Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2015

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München

Programm heute

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

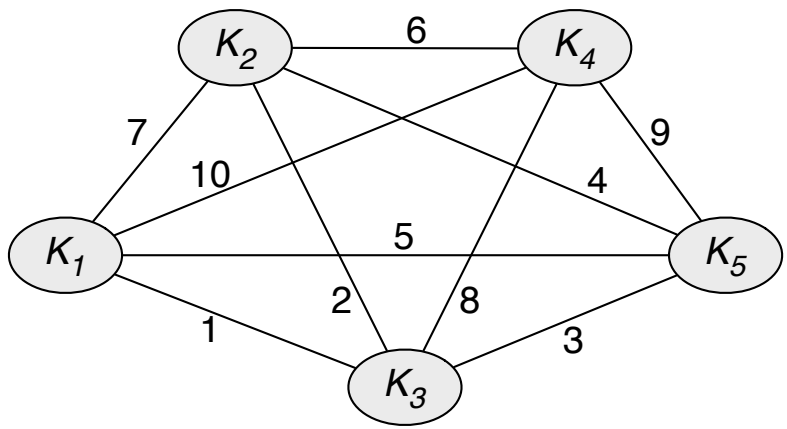
Übersicht: Graphen

Klassen von Graphen:

- ungerichteter Graph
- gerichteter Graph

Zusätzliche Eigenschaften:

- gewichteter Graph



Definition: Ungerichteter Graph

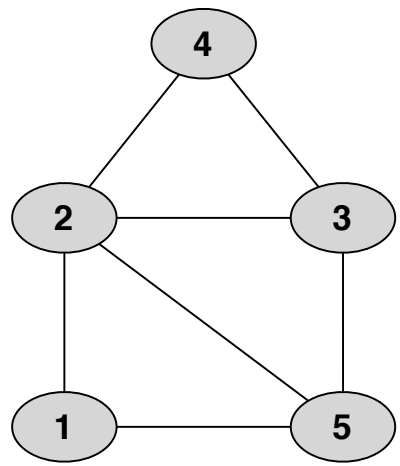
Definition: Ungerichteter Graph

Ein ungerichteter Graph ist ein Paar $G = (V, E)$ mit

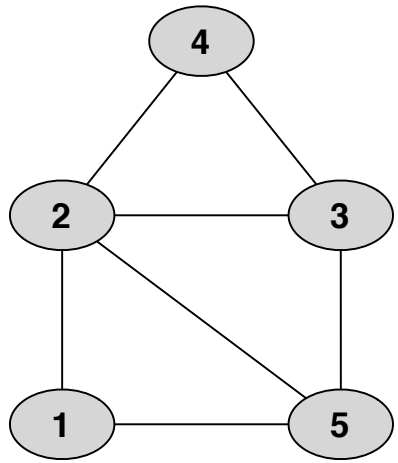
- V endliche Menge der Knoten
- $E \subseteq \big\{\{u, v\} : \ u, v \in V\big\}$ Menge der Kanten

- auf Englisch:
  - Knoten = vertices
  - Kanten = edges
- es ist $\{u, v\} = \{v, u\}$, d.h. Richtung der Kante spielt keine Rolle

Beispiel:



Ungerichteter Graph: Beispiel



- Graph $G_u = (V_u, E_u)$
- Knoten $V_u = \{1, 2, 3, 4, 5\}$
- Kanten
  $E_u = \big\{\{1, 2\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}\big\}$

Definition: Gerichteter Graph

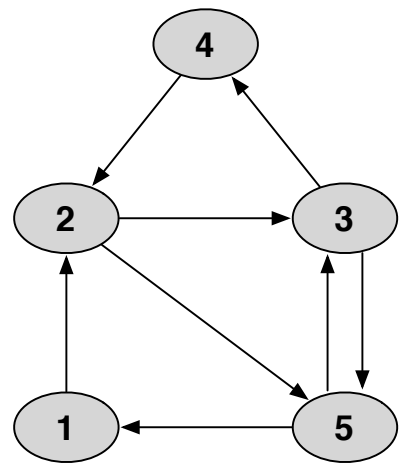
Definition: Gerichteter Graph

Ein gerichteter Graph ist ein Paar $G = (V, E)$ mit

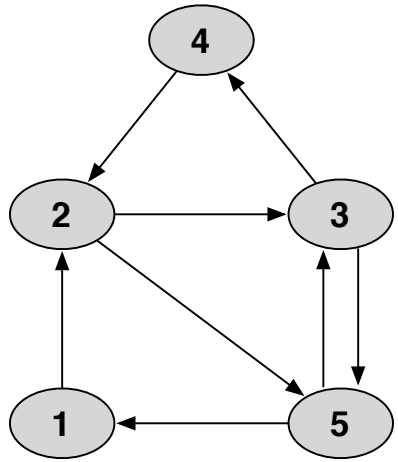
- V endliche Menge der Knoten
- $E \subseteq V \times V$ Menge der Kanten

- $E \subseteq \{(u, v) : \ u, v \in V\} = V \times V$
- es ist $(u, v) \neq (v, u)$, d.h. Richtung der Kante spielt eine Rolle
- hier sind Schleifen möglich, d.h. Kanten der Form (u, u) für $u \in V$

Beispiel:



Gerichteter Graph: Beispiel



- Graph $G_g = (V_g, E_g)$
- Knoten $V_g = \{1, 2, 3, 4, 5\}$
- Kanten
  $E_g = \{(1,2), (2,3), (2,5), (3,4), (3,5), (4,2), (5,1), (5,3)\}$

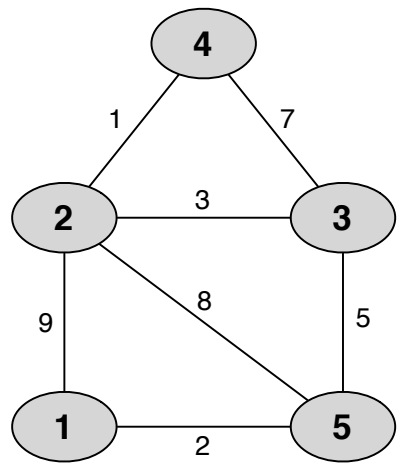
Definition: Gewichteter Graph

Definition: Gewichteter Graph

Ein gewichteter Graph ist ein Graph $G = (V, E)$ mit einer Gewichtsfunktion $w : E \to \mathbb{R}$.

- der Graph G kann gerichtet oder ungerichtet sein
- je nach Anwendung kann ein verschiedener Wertebereich für die Funktion w gewählt werden
  - z.B. $\mathbb{R}$ oder $\mathbb{N}_0$

Beispiel:



Eigenschaften von Graphen I

Sei $G = (V, E)$ ein Graph (gerichtet oder ungerichtet).

- Ist $(u, v) \in E$ bzw. $\{u, v\} \in E$ für $u, v \in V$, so heißt v adjazent zu u.
- Sei G gerichteter Graph, sei $v \in V$ ein Knoten.
  - die Anzahl der eintretenden Kanten in v heißt Eingangsgrad (englisch: indegree) von v,

$$indeg(v) = \left|\left\{v' : \ (v', v) \in E\right\}\right|$$

  - die Anzahl der austretetenden Kanten von v heißt Ausgangsgrad (englisch: outdegree) von v,

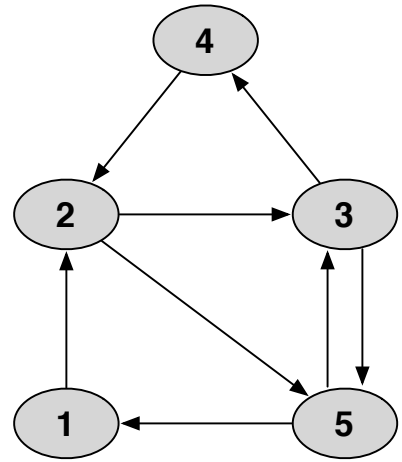
$$outdeg(v) = \left|\left\{v' : \ (v, v') \in E\right\}\right|$$

- Sei G ungerichteter Graph, sei $v \in V$ ein Knoten.
  - die Anzahl der eintretenden bzw. austretenden Kanten von v heißt Grad (englisch: degree) von v oder kurz $deg(v)$.

Eigenschaften von Graphen: Beispiel

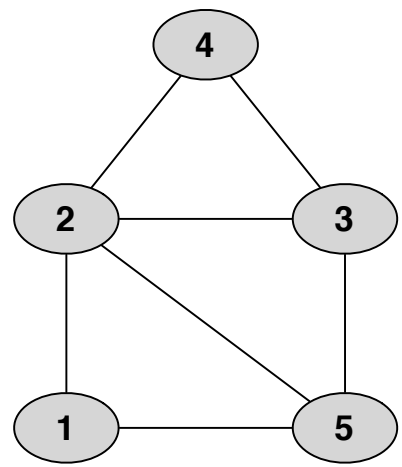
Beispiel gerichteter Graph:

- Knoten 2 ist adjazent zu Knoten 1
- Knoten 1 ist adjazent zu Knoten 5
- $outdeg(2) = 2$, $indeg(2) = 2$
- $indeg(1) = 1$, $outdeg(1) = 1$



Beispiel ungerichteter Graph:

- Knoten 4 ist adjazent zu Knoten 2
- Knoten 2 ist adjazent zu Knoten 4
- $deg(2) = 4$
- $deg(1) = 2$



Eigenschaften von Graphen II

Sei $G = (V, E)$ ein Graph (gerichtet oder ungerichtet).

- Seien $v, v' \in V$. Ein Pfad von v nach v' ist eine Folge von Knoten $(v_0, v_1, \ldots, v_k) \subset V$ mit
  - $v_0 = v$, $v_k = v'$
  - $(v_i, v_{i+1}) \in E$ bzw. $\{v_i, v_{i+1}\} \in E$ für $i = 0, \ldots, k-1$

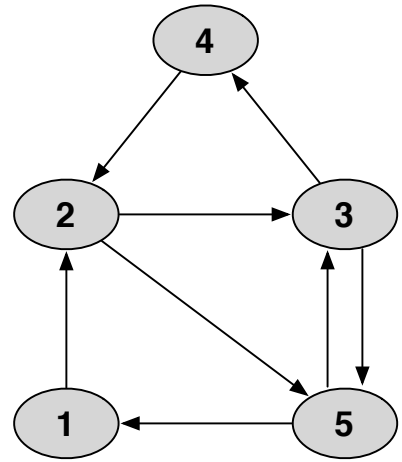
  k heißt Länge des Pfades.
- Ein Pfad heißt einfach, falls alle Knoten des Pfades paarweise verschieden sind.

Gibt es einen Pfad von u nach v, so heißt v erreichbar von u.

Eigenschaften von Graphen II: Beispiel

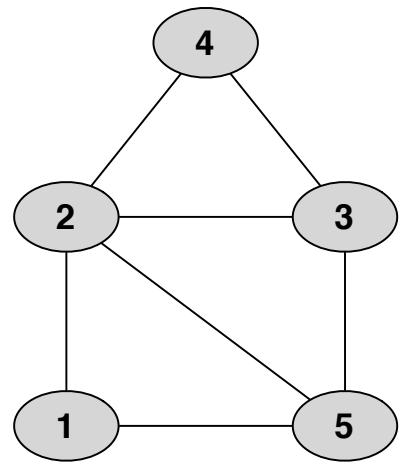
Beispiel gerichteter Graph:

- $(1, 2, 5)$ ist ein einfacher Pfad der Länge 2
- $(2, 3, 4, 2)$ ist ein Pfad der Länge 3, aber nicht einfach
- 5 ist erreichbar von 1



Beispiel ungerichteter Graph:

- $(5, 2, 4, 3, 2)$ ist ein Pfad der Länge 4, aber nicht einfach
- $(1, 2, 3, 4)$ ist ein einfacher Pfad der Länge 3
- 3 ist erreichbar von 1



Eigenschaften von Graphen III

Sei $G = (V, E)$ gerichteter Graph.

- Ein Pfad $(v_0, \ldots, v_k)$ heißt Zyklus, falls
  - $v_0 = v_k$ und
  - der Pfad mindestens eine Kante enthält
- Ein Zyklus $(v_0, \ldots, v_k)$ heißt einfach, falls $v_1, \ldots, v_k$ paarweise verschieden sind.

Sei $G = (V, E)$ ungerichteter Graph.

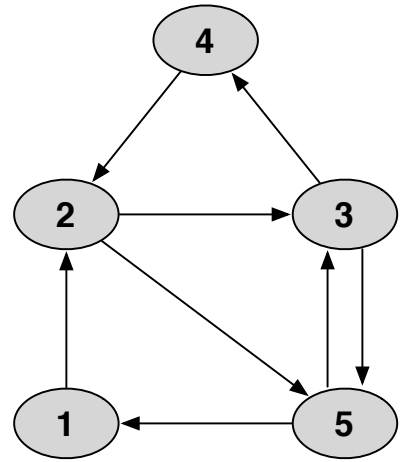
- Ein Pfad $(v_0, \ldots, v_k)$ heißt Zyklus, falls
  - $v_0 = v_k$
  - $k \geq 3$
  - $v_1, \ldots, v_k$ paarweise verschieden

Ein Graph ohne Zyklen heißt azyklisch.

Eigenschaften von Graphen III: Beispiel

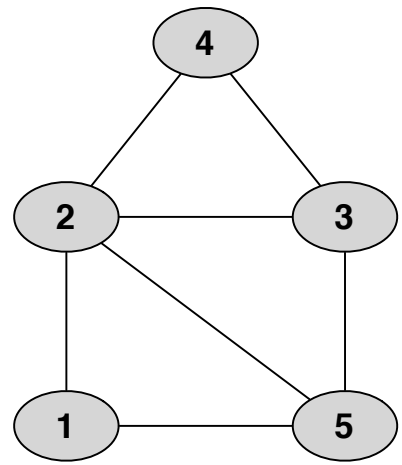
Beispiel gerichteter Graph:

- $(1, 2, 5, 3, 5, 1)$ ist ein Zyklus, aber nicht einfach
- $(1, 2, 5, 1)$ ist ein einfacher Zyklus



Beispiel ungerichteter Graph:

- $(1, 2, 5, 1)$ ist ein Zyklus
- $(2, 3, 4, 2)$ ist ein Zyklus



Eigenschaften von Graphen IV

Sei $G = (V, E)$ gerichteter Graph.

- G heißt stark zusammenhängend, falls jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine starke Zusammenhangskomponente von G ist ein maximaler zusammenhängender Untergraph von G.
  - alternativ: Äquivalenzklassen der Knoten bezüglich Relation "gegenseitig erreichbar"

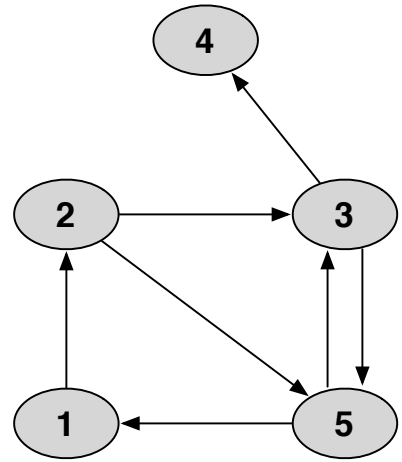
Sei $G = (V, E)$ ungerichteter Graph.

- G heißt zusammenhängend, falls jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- Eine Zusammenhangskomponente von G ist ein maximaler zusammenhängender Untergraph von G.
  - alternativ: Äquivalenzklassen der Knoten bezüglich Relation "erreichbar von"

Eigenschaften von Graphen IV: Beispiel

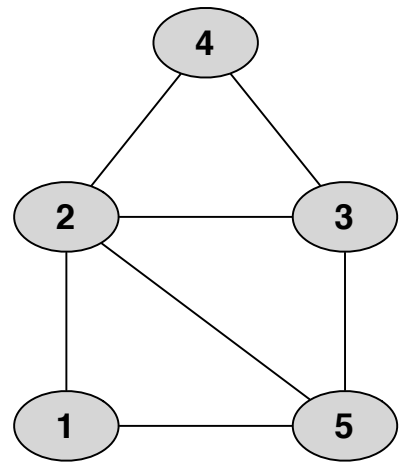
Beispiel gerichteter Graph:

- Graph ist nicht stark zusammenhängend (z.B. 3 nicht erreichbar von 4)
- starke Zusammenhangskomponenten: $\{1, 2, 3, 5\}$ und $\{4\}$



Beispiel ungerichteter Graph:

- Graph ist zusammenhängend
- nur eine Zusammenhangskomponente: $\{1, 2, 3, 4, 5\}$



Darstellung von Graphen: Adjazenzmatrizen

Adjazenzmatrix

Sei $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$. Die Adjazenzmatrix von G speichert die vorhandenen Kanten in einer $n \times n$ Matrix $A \in \mathbb{R}^{n \times n}$ mit

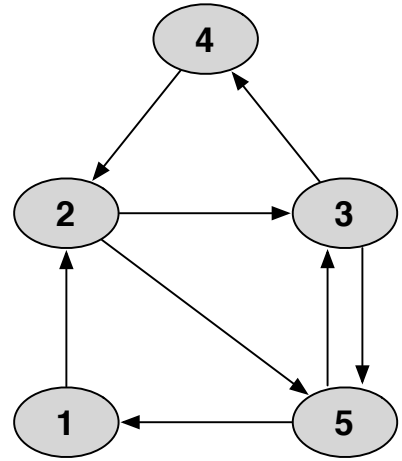
- $A(i, j) = 1$ falls Kante von Knoten v_i zu v_j existiert
- $A(i, j) = 0$ falls keine Kante von Knoten v_i zu v_j existiert

für $i, j \in \{1, \ldots, n\}$.

Adjazenzmatrizen: Beispiele

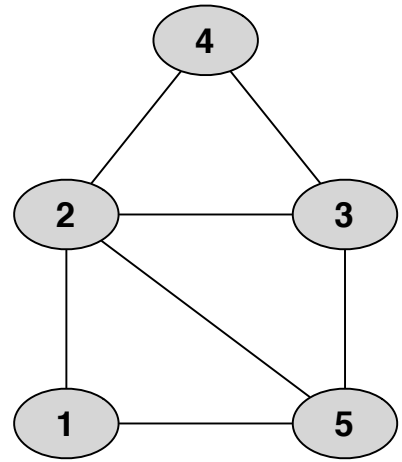
Beispiel gerichteter Graph:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$



Beispiel ungerichteter Graph:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$



Adjazenzmatrizen: Eigenschaften

Eigenschaften von Adjazenzmatrizen zu Graph $G = (V, E)$

- sinnvoll wenn der Graph nahezu vollständig ist (d.h. fast alle möglichen Kanten tatsächlich in E liegen)
- Speicherkomplexität: $O(|V|^2)$
- bei ungerichteten Graphen ist die Adjazenzmatrix symmetrisch
- bei gewichteten Graphen kann man statt der 1 in der Matrix das Gewicht der Kante eintragen

Darstellung von Graphen: Adjazenzlisten

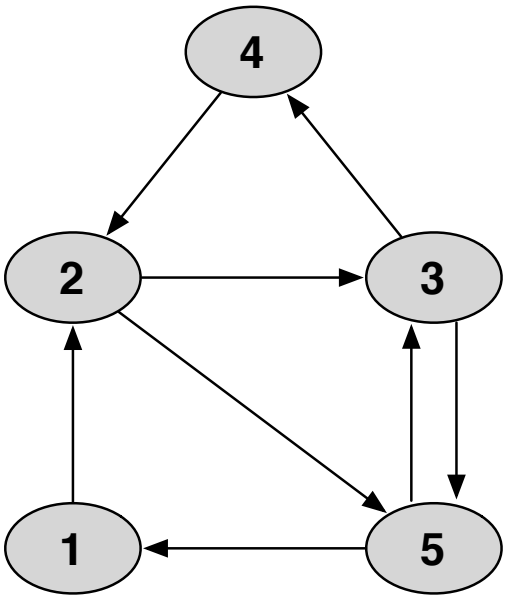
Adjazenzliste

Sei $G = (V, E)$ gerichteter Graph. Eine Adjazenzliste von G sind $|V| + 1$ verkettete Listen, so daß

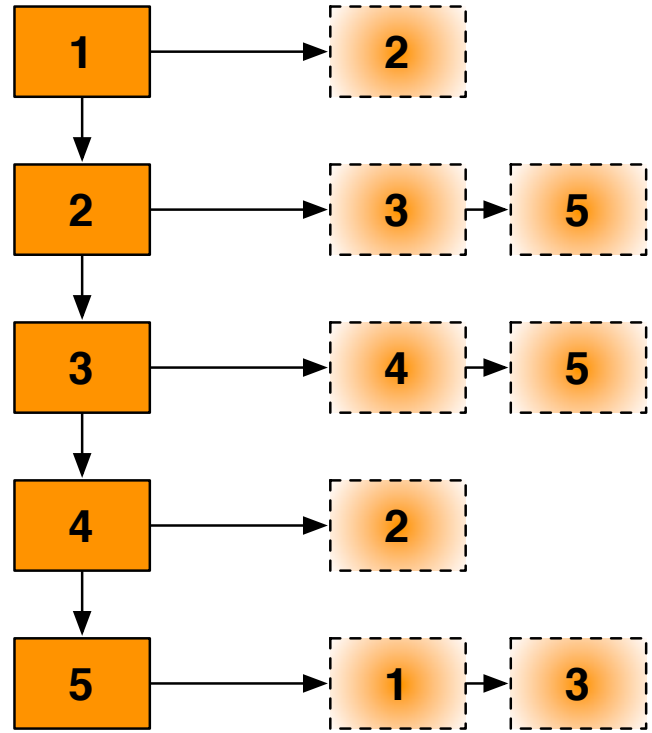
- die erste Liste alle Knoten enthält
- für jeden Knoten v eine Liste angelegt wird mit allen Knoten, die durch eine von v austretende Kante zu erreichen sind

Adjazenzliste: Beispiel

Graph:



Adjazenzliste:



Adjazenzliste: Eigenschaften

Eigenschaften von Adjazenzlisten zu Graph $G = (V, E)$

- sinnvoll bei dünn besetzten Graphen mit wenigen Kanten
- Speicherkomplexität: $O(|V| + |E|)$
- bei ungerichteten Graphen gleiches Verfahren
  - allerdings muß jede Kante zweimal gespeichert werden
- bei gewichteten Graphen kann man die Gewichte mit in den verketteten Listen der jeweiligen Knoten speichern

Komplexität der Darstellungen

Sei $G = (V, E)$ Graph.

Operation	Adjazenzmatrix	Adjazenzliste
Kante einfügen	$O(1)$	$O(\|V\|)$
Kante löschen	$O(1)$	$O(\|V\|)$
Knoten einfügen	$O(\|V\|^2)$	$O(1)$
Knoten löschen	$O(\|V\|^2)$	$O(\|V\| + \|E\|)$

- falls Größe im Vorhinein bekannt, kann Knoten löschen/einfügen bei Adjanzenzmatrix effizienter implementiert werden
- auch die Adjanzenzlisten lassen sich effizienter implementieren, z.B. auf Kosten von Speicher (sequentielle Liste statt verkettete Liste)
- Löschen von Knoten ist immer aufwendig, da auch alle Kanten von/zu diesem Knoten gelöscht werden müssen

Algorithmen auf Graphen

Ausblick auf Algorithmen auf Graphen:

- Traversierung (Durchlaufen) von allen Knoten
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)
- kürzester Pfad zwischen Knoten in Graphen
- minimaler Spannbaum (minimum spanning tree, MST)

→ siehe Kapitel 9!

Programm heute

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

Bäume

Bäume sind alltägliches Mittel zur Strukturierung:

- Stammbaum
- Hierarchie in Unternehmen
- Systematik in der Biologie
- etc.

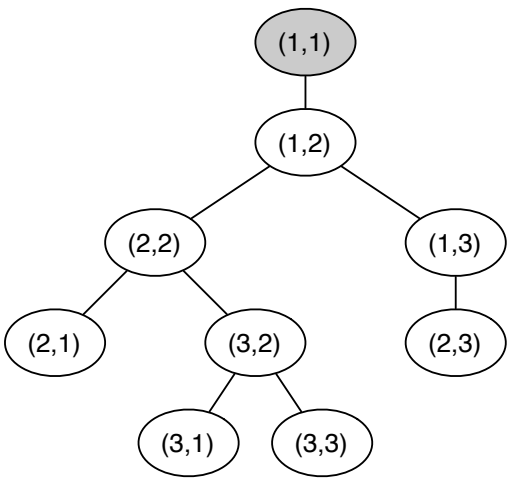
In Informatik:

- Bäume sind spezielle Graphen
- Wurzel oben!

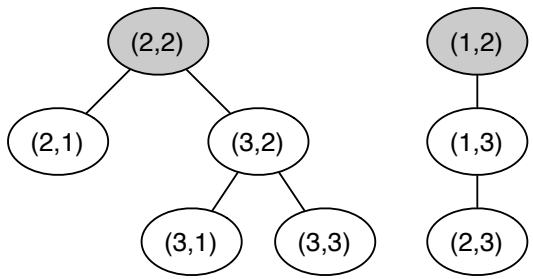
Definition Wald/Baum

Definition: Wald und Baum

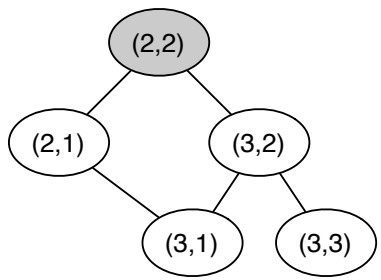
- Ein azyklischer ungerichteter Graph heißt auch Wald.
- Ein zusammenhängender, azyklischer ungerichteter Graph heißt auch Baum.



Baum



Wald



kein Baum!

Eigenschaften von Bäumen

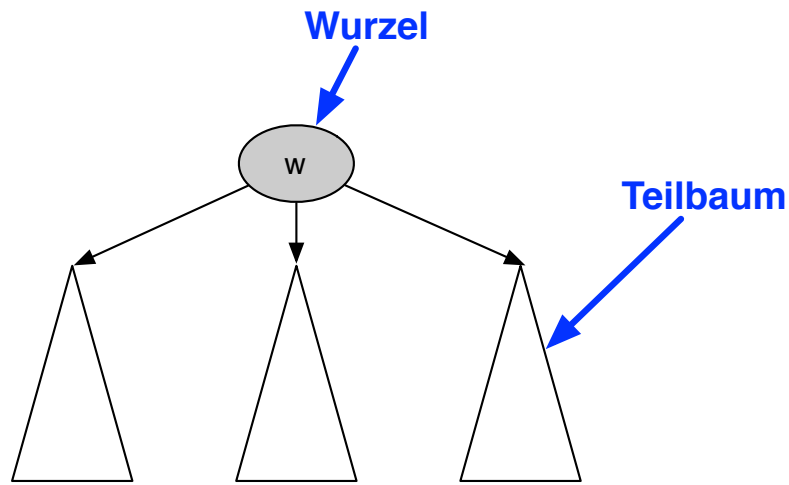
Sei $G = (V, E)$ ein Baum.

- jedes Paar von Knoten $u, v \in V$ ist durch einen einzigen Pfad verbunden
- G ist zusammenhängend, aber wenn eine Kante aus E entfernt wird, ist G nicht mehr zusammenhängend
- G ist azyklisch, aber wenn eine Kante zu E hinzugefügt wird, ist G nicht mehr azyklisch
- es ist $|E| = |V| - 1$

Wurzel von Bäumen

Sei $G = (V, E)$ ein Baum.

- genau ein Knoten $w \in V$ wird als Wurzel ausgezeichnet
- entfernt man w erhält man einen Wald von Teilbäumen

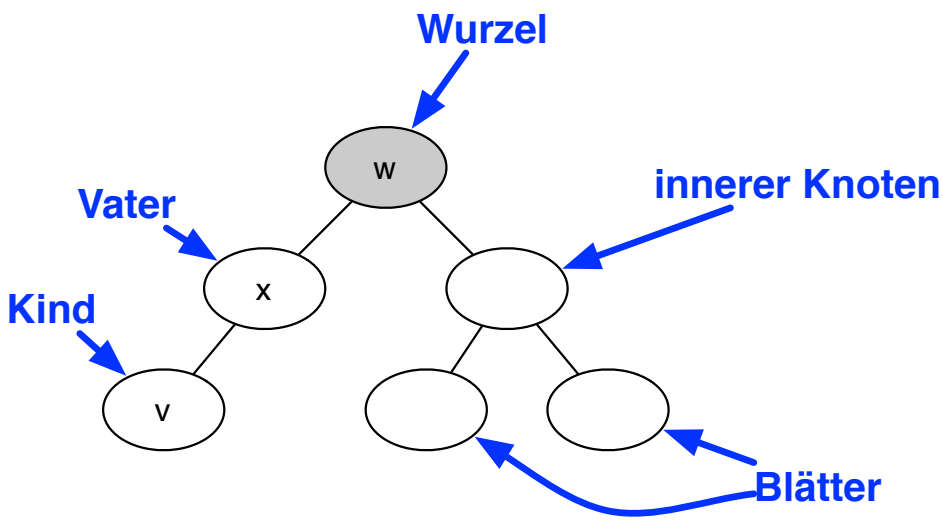


Hinweis: manchmal wird zwischen "freiem" und "gewurzeltem" Baum unterschieden!

Weitere Begriffe bei Bäumen I

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

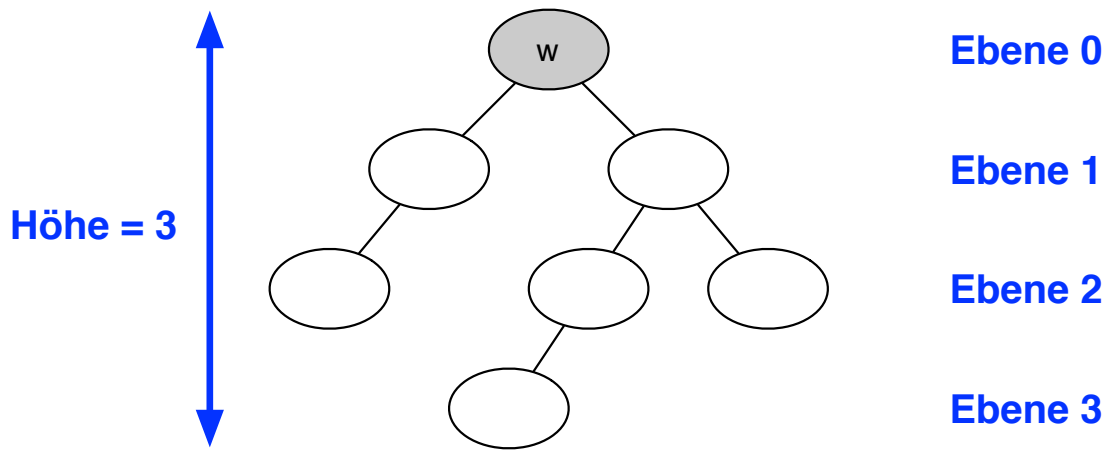
- jeder Knoten $v \in V$ mit $v \neq w$ ist mit genau einer Kante mit seinem Vaterknoten $x \in V$ (oder: Vorgänger) verbunden
- v wird dann als Kind (oder: Sohn, Nachfolger) von $x \in V$ bezeichnet
- ein Knoten ohne Kinder heißt Blatt, alle anderen Knoten heißen innere Knoten



Weitere Begriffe bei Bäumen II

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

- Anzahl der Kinder von Knoten $x \in V$ heißt auch Grad von x
  - Achtung: der Grad in dem ungerichteten Graph G ist anders definiert!
- Länge des Pfades von Wurzel w zu Knoten $x \in V$ heißt Tiefe von x
- alle Knoten gleicher Tiefe bilden eine Ebene des Baumes G
- die Höhe des Baumes G ist die maximale Tiefe der Knoten von G
  - Achtung: manchmal ist Höhe auch als Anzahl der Ebenen definiert

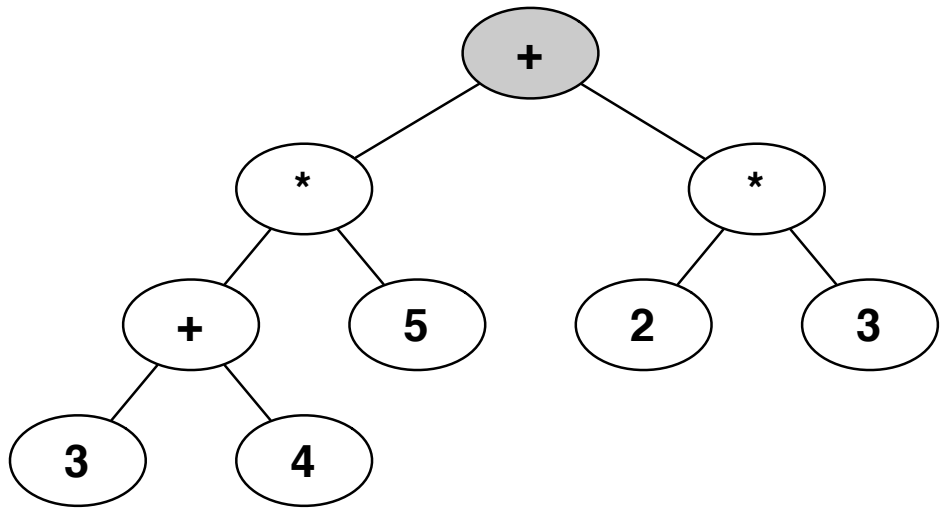


Bäume: Beispiele

- arithmetischer Ausdruck

$$(3 + 4) * 5 + 2 * 3$$

repräsentiert als Baum:



- hierarchisches Dateisystem
  - Windows z.B. "C:\"
  - Unix "/"
- Suchbaum → Kapitel 8

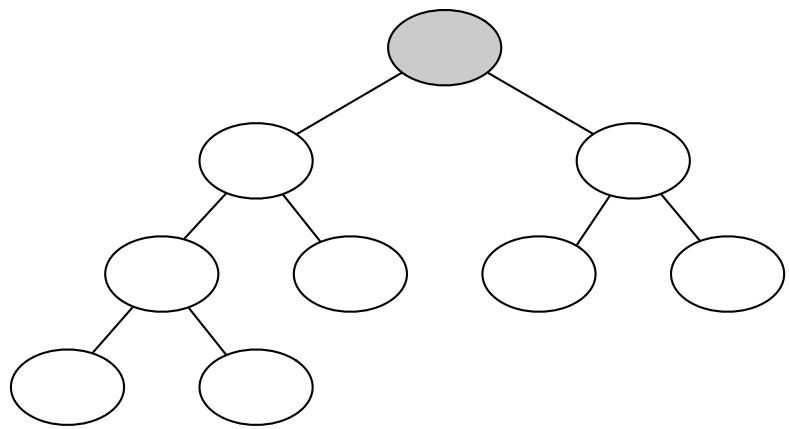
Besondere Bäume

Sei $G = (V, E)$ ein Baum mit Wurzel $w \in V$.

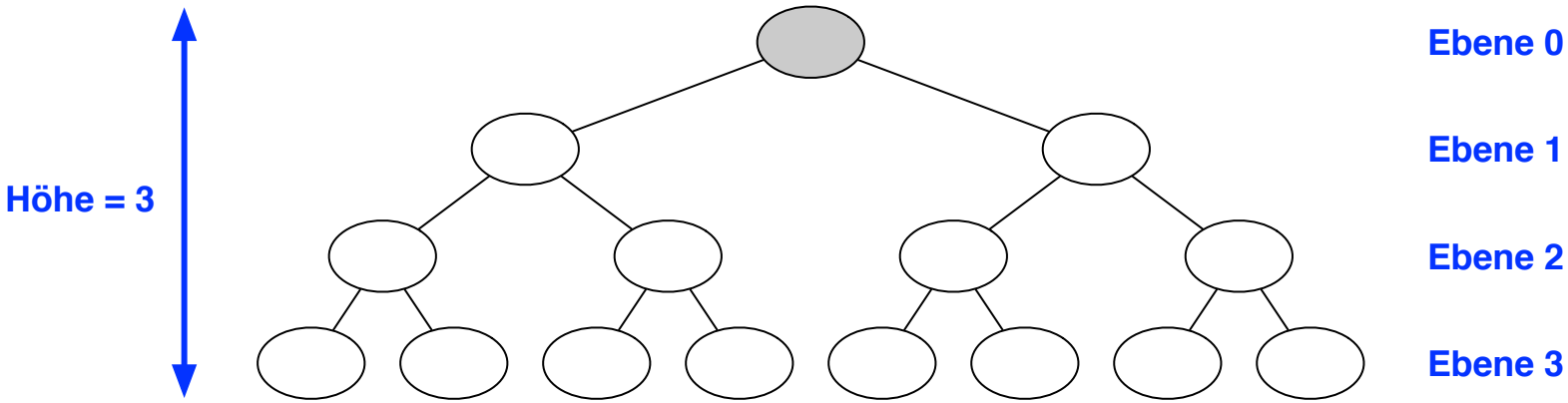
- sind die Kinder jedes Knotens in bestimmter Reihenfolge angeordnet, heißt G geordneter Baum
- ist die Anzahl n der Kinder jedes Knotens vorgegeben, heißt G n-ärer Baum

Wichtiger Spezialfall:

- ist G geordnet und hat jeder Knoten maximal zwei Kinder, heißt G Binärbaum



Beispiel: Binärbaum



Binärbaum mit Höhe 3, 8 Blättern und 7 inneren Knoten.

- Binärbaum heißt vollständig, wenn jede Ebene die maximale Anzahl an Knoten enthält
- ein vollständiger Binärbaum der Höhe k hat $2^{k+1} - 1$ Knoten, davon 2^k Blätter

Darstellung von Bäumen: als Graph

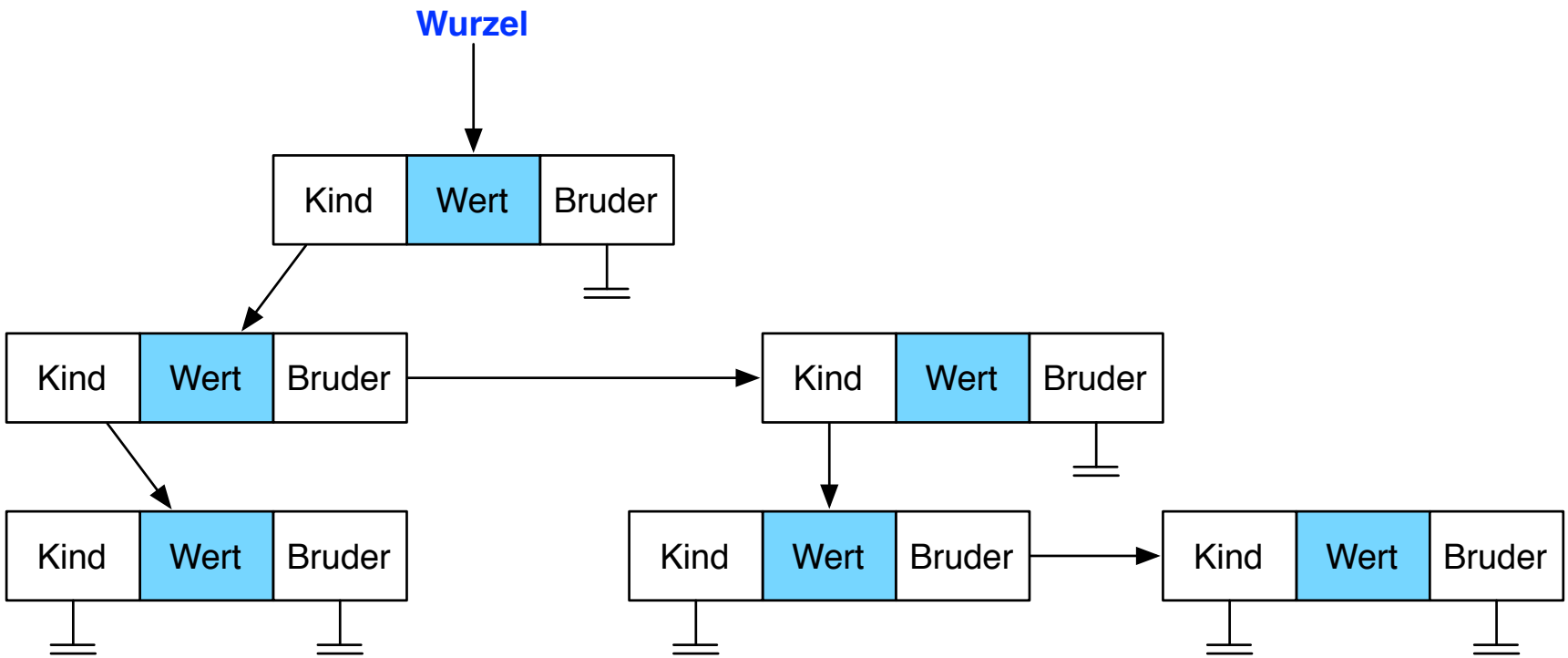
Bäume sind Graphen $\rightarrow$ Darstellung als

- Adjazenzmatrix
- Adjazenzliste

$\rightarrow$ leider meist nicht effizient (sowohl Laufzeit als auch Speicher)

Darstellung von Bäumen: verkettete Liste

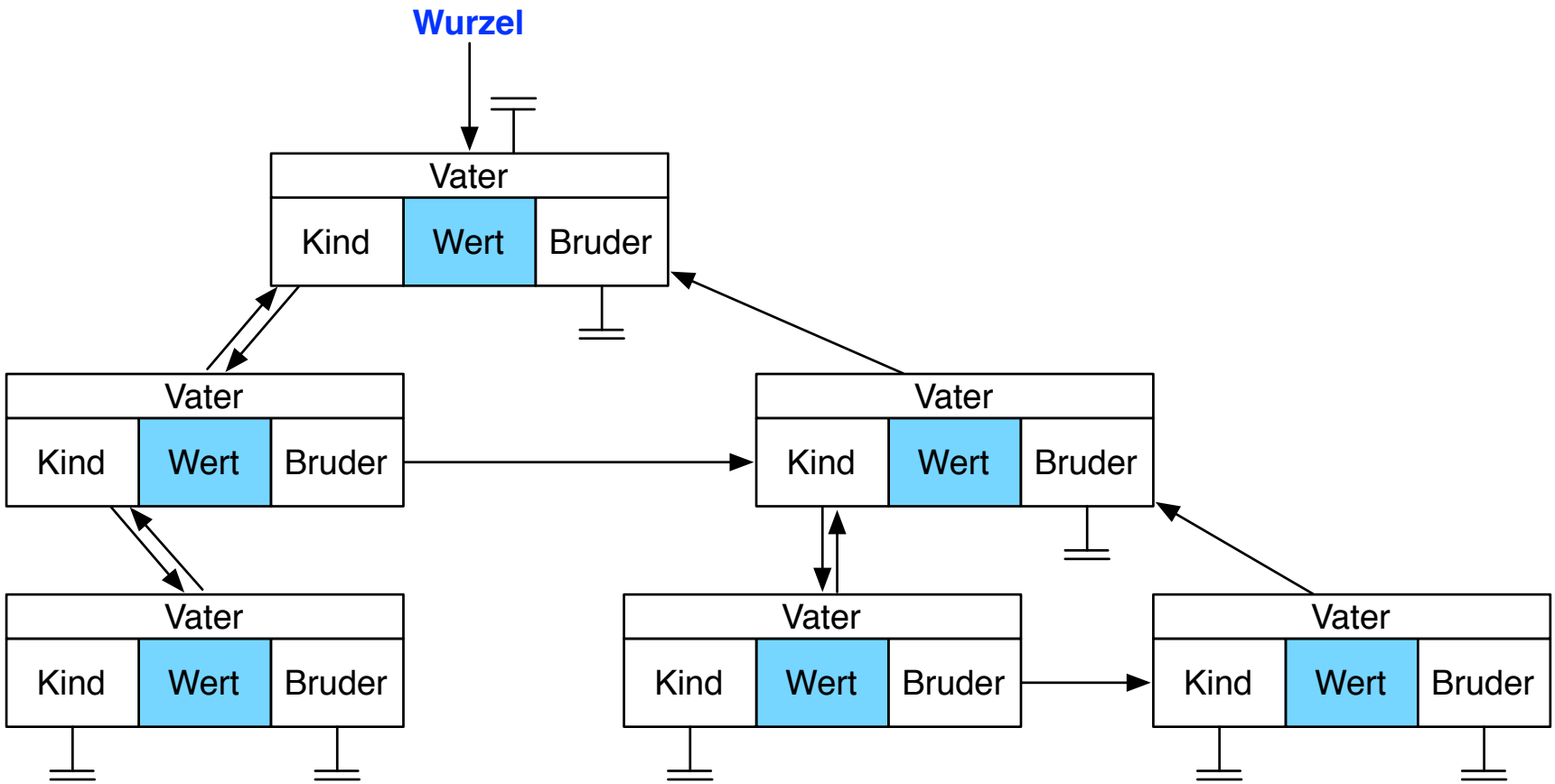
Darstellung mit angepassten verketteten Listen:



Nachteil: nur Navigation “nach unten” möglich

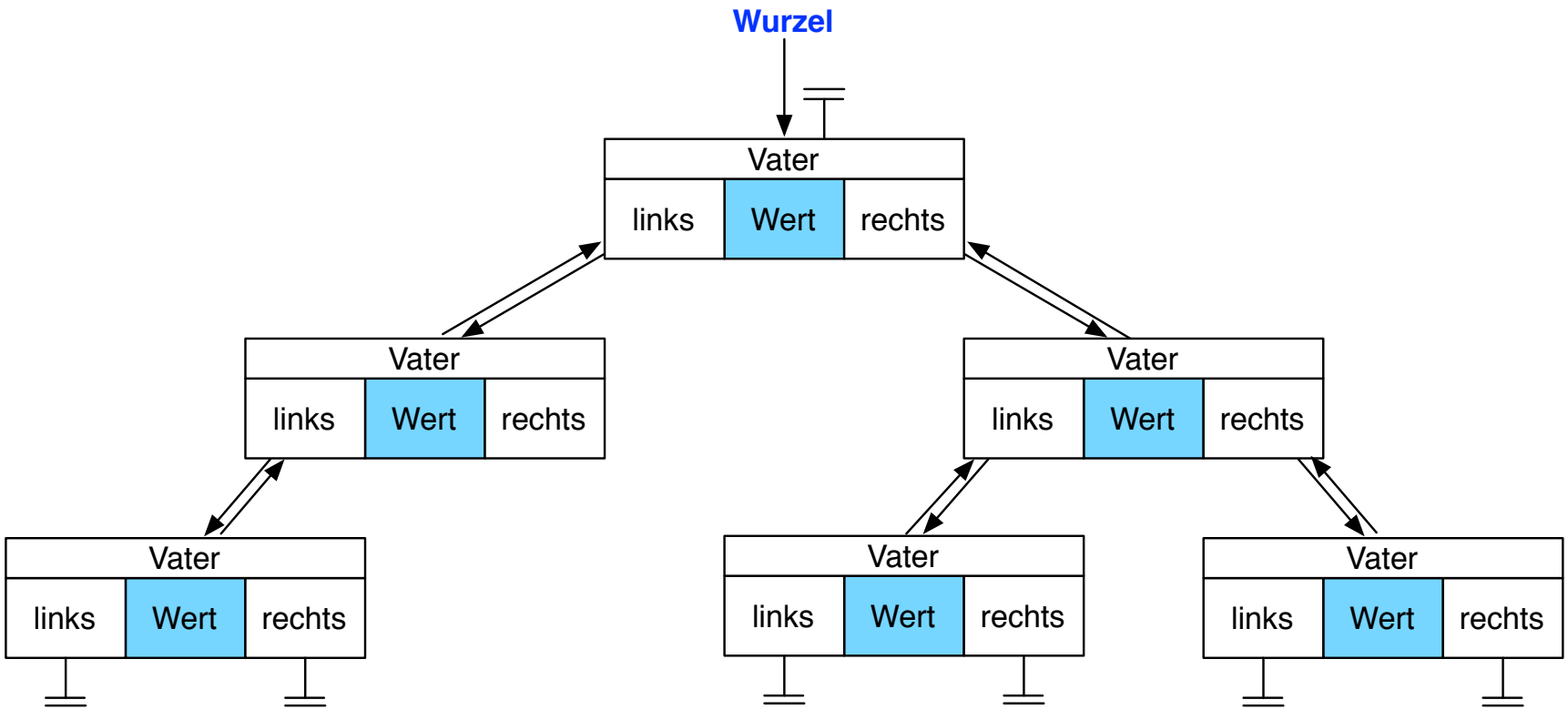
Darstellung von Bäumen: doppelt verkettete Liste

Darstellung mit angepassten doppelt verketteten Listen:



Darstellung von Binärbäumen

Darstellung direkt mit linkem/rechtem Kind (doppelt verkettet):



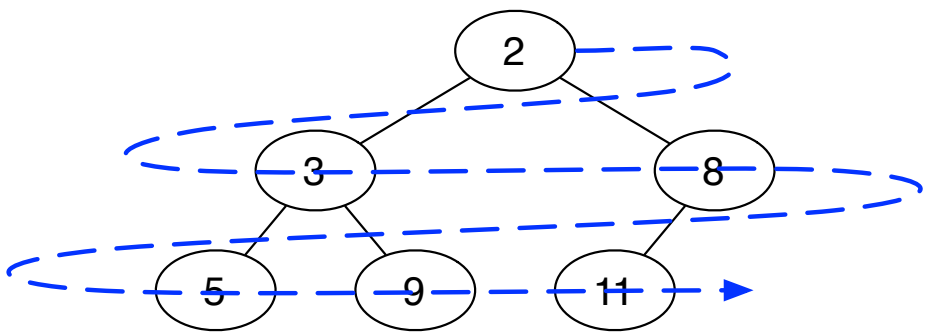
Binärbaum als sequentielle Liste I

- vollständiger Binärbaum Höhe k hat $2^{k+1} - 1$ Knoten

  $\rightarrow$ speichere Knoten von oben nach unten, von links nach rechts in sequentieller Liste (Array)

  $\rightarrow$ maximale Grösse von Array: $2^{k+1} - 1$

- Beispiel fast vollständiger Binärbaum:



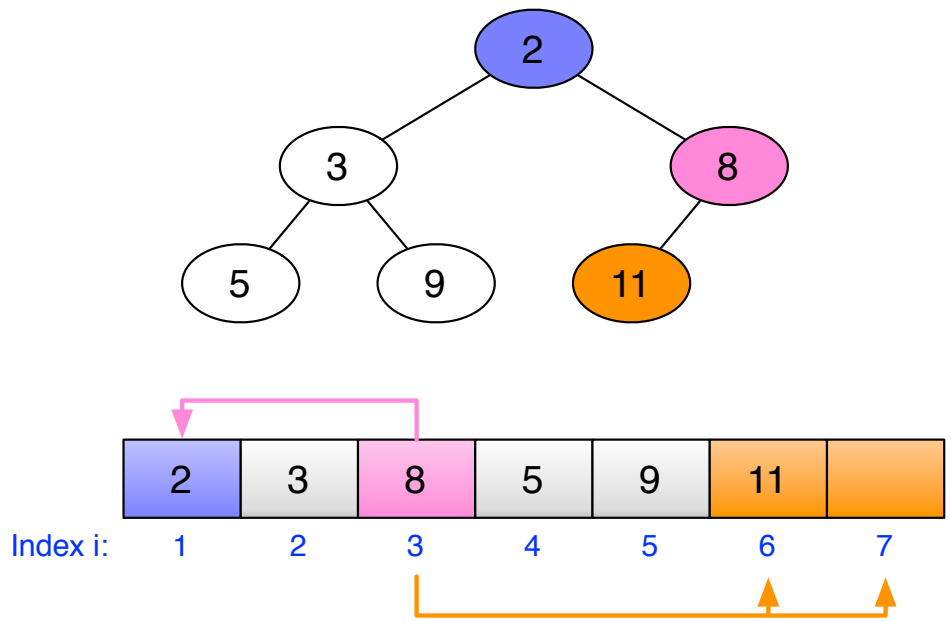
2	3	8	5	9	11	

Binärbaum als sequentielle Liste II

- Wurzel: an Position 1
- Knoten an Position i:
  - Vater-Knoten an Position $\lfloor i/2 \rfloor$
  - linkes Kind an Position $2i$;
  - rechtes Kind an Position $2i + 1$

Pseudocode:

- **vater**(i): **return** $\lfloor i/2 \rfloor$;
- **links**(i): **return** $2i$;
- **rechts**(i): **return** $2i + 1$;

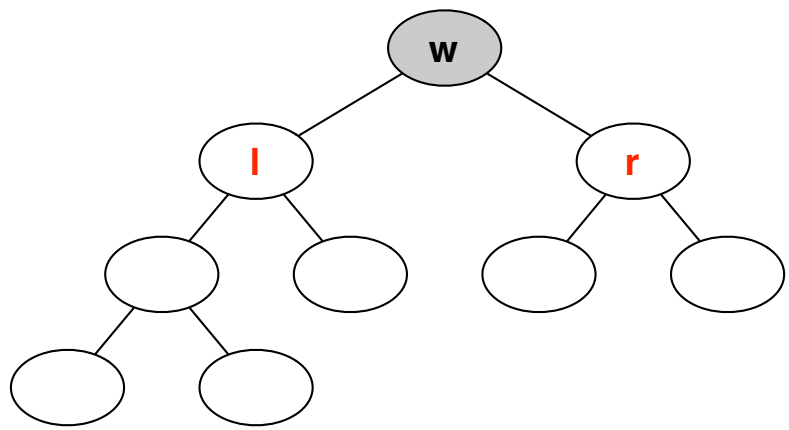


Traversierung von Binärbäumen

Sei $G = (V, E)$ Binärbaum.

In welcher Reihenfolge durchläuft man G?

- Wurzel zuerst
- danach linker oder rechter Kind-Knoten l bzw. r?
- falls l: danach Kind-Knoten von l oder zuerst r?
- falls r: danach Kind-Knoten von r oder zuerst l?



$\rightarrow$ falls zuerst in die Tiefe: Depth-first search (DFS)

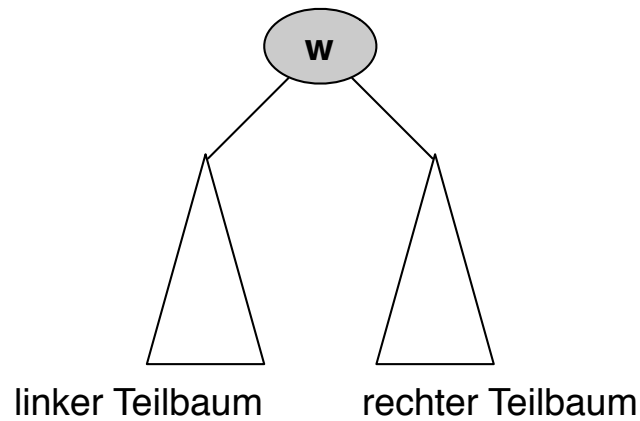
$\rightarrow$ falls zuerst in die Breite: Breadth-first search (BFS)

DFS Binärbaum

Sei $G = (V, E)$ Binärbaum.

Tiefensuche (Depth-first search, DFS) gibt es in 3 Varianten:

1. Pre-order Reihenfolge
   - besuche Wurzel
   - durchlaufe linken Teilbaum
   - durchlaufe rechten Teilbaum
2. In-order Reihenfolge
   - durchlaufe linken Teilbaum
   - besuche Wurzel
   - durchlaufe rechten Teilbaum
3. Post-order Reihenfolge
   - durchlaufe linken Teilbaum
   - durchlaufe rechten Teilbaum
   - besuche Wurzel



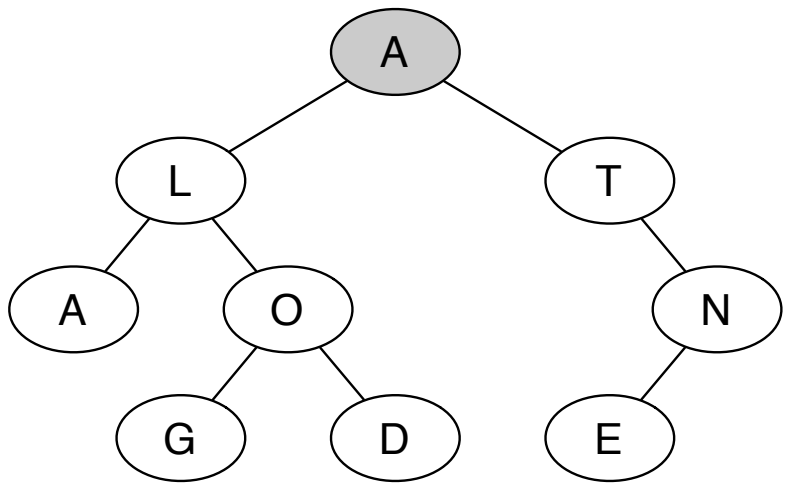
Pre-order Traversierung

Pre-order Reihenfolge:

- besuche Wurzel
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum

Beispiel:

A, L, A, O, G, D, T, N, E



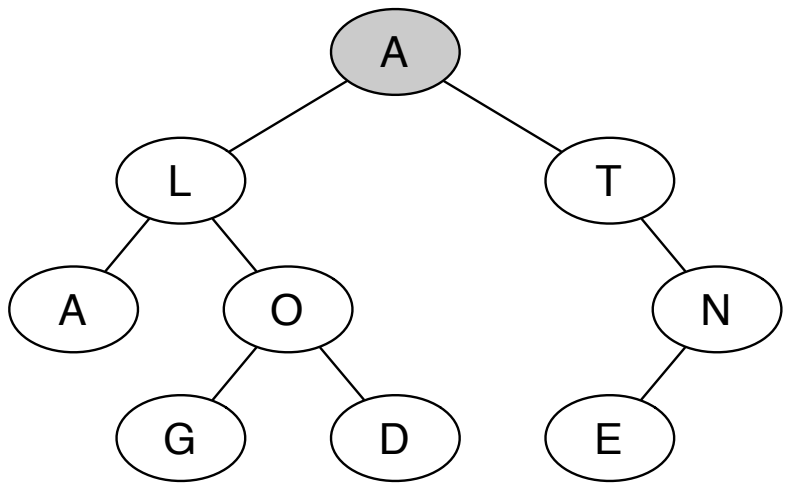
In-order Traversierung

In-order Reihenfolge:

- durchlaufe linken Teilbaum
- besuche Wurzel
- durchlaufe rechten Teilbaum

Beispiel:

A, L, G, O, D, A, T, E, N



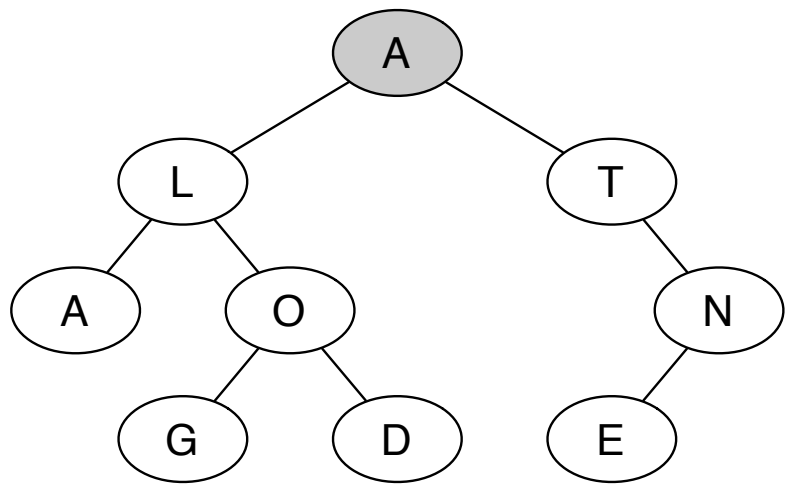
Post-order Traversierung

Post-order Reihenfolge:

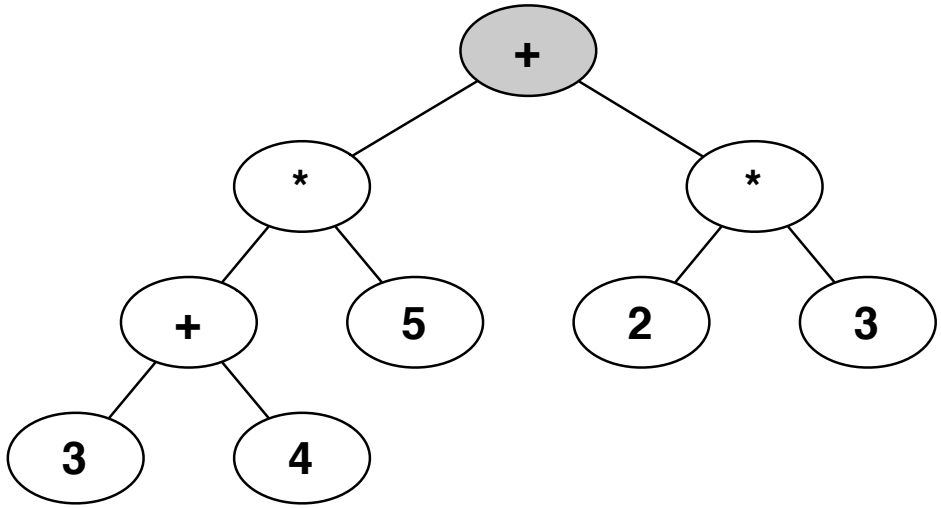
- durchlaufe linken Teilbaum
- durchlaufe rechten Teilbaum
- besuche Wurzel

Beispiel:

A, G, D, O, L, E, N, T, A



Beispiel: Arithmetischer Term



Traversierung:

- Pre-order: + * + 3 4 5 * 2 3
- In-order: 3 + 4 * 5 + 2 * 3
- Post-order: 3 4 + 5 * 2 3 * +

Implementierung DFS Traversierung

Datenstruktur:

- Algorithmus preorder:

```
preorder(knoten):
    if (knoten == null) return;
    besuche(knoten);
    preorder(knoten.links);
    preorder(knoten.rechts);
```

- Algorithmus inorder:

```
inorder(knoten):
    if (knoten == null) return;
    inorder(knoten.links);
    besuche(knoten);
    inorder(knoten.rechts);
```

- Algorithmus postorder:

```
postorder(knoten):
    if (knoten == null) return;
    postorder(knoten.links);
    postorder(knoten.rechts);
    besuche(knoten);
```

- rekursive Algorithmen
- auf Call-Stack basiert

Implementierung DFS Traversierung ohne Rekursion

- Datenstruktur:

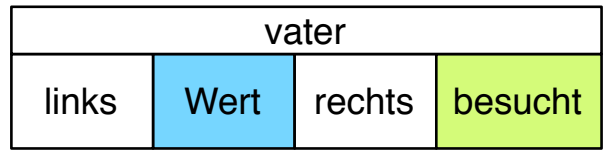
- Hilfsmittel: Stack von Knoten “knotenStack”

```
postorderIterativ(wurzelKnoten):
    knotenStack.push(wurzelKnoten);
    while ( !knotenStack.isEmpty() ) {
       knoten = knotenStack.top();
       if ( (knoten.links != null) && (!knoten.links.besucht) )
          knotenStack.push(knoten.links);
       else if ( (knoten.rechts != null) && (!knoten.rechts.besucht) )
          knotenStack.push(knoten.rechts);
       else {
          besuche(knoten);
          knoten.besucht = true;
          knotenStack.pop();
       }
    }
```

Implementierung ohne Rekursion und ohne Stack

- Datenstruktur:

Knoten:



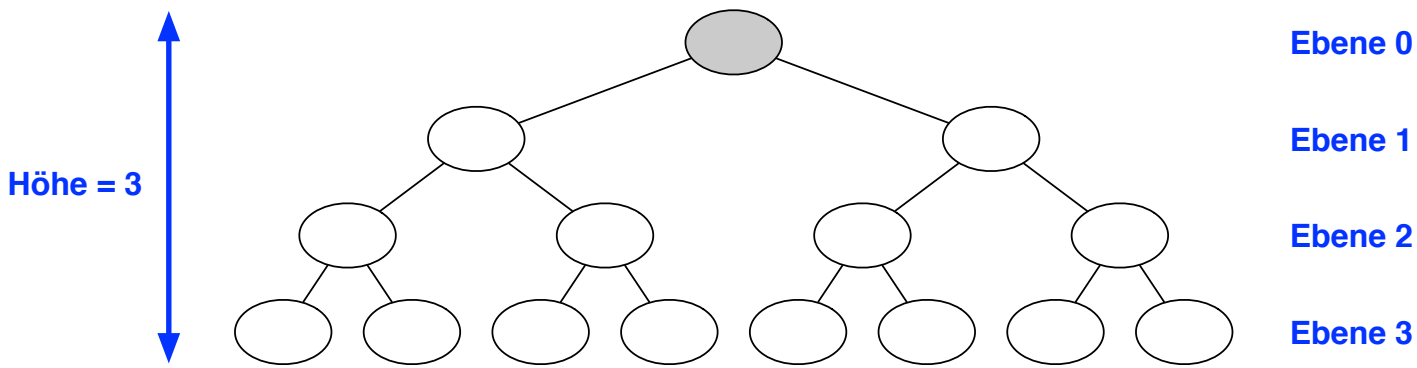

```
postorderIterativeOhneStack(wurzelKnoten):
    knoten = wurzelKnoten;
    while (true) {
      if ( (knoten.links != null) && (!knoten.links.besucht) )
        knoten = knoten.links;
      else if ( (knoten.rechts != null) && (!knoten.rechts.besucht) )
        knoten = knoten.rechts;
      else {
        besuche(knoten);
        knoten.besucht = true;
        if (knoten.vater == null) break;
        else knoten = knoten.vater;
      }
    }
```

BFS Binärbaum

Sei $G = (V, E)$ Binärbaum.

Breitensuche (Breadth-first search, BFS):

- besuche Wurzel
- für alle Ebenen von 1 bis Höhe
  - besuche alle Knoten aktueller Ebene



Implementierung BFS Traversierung

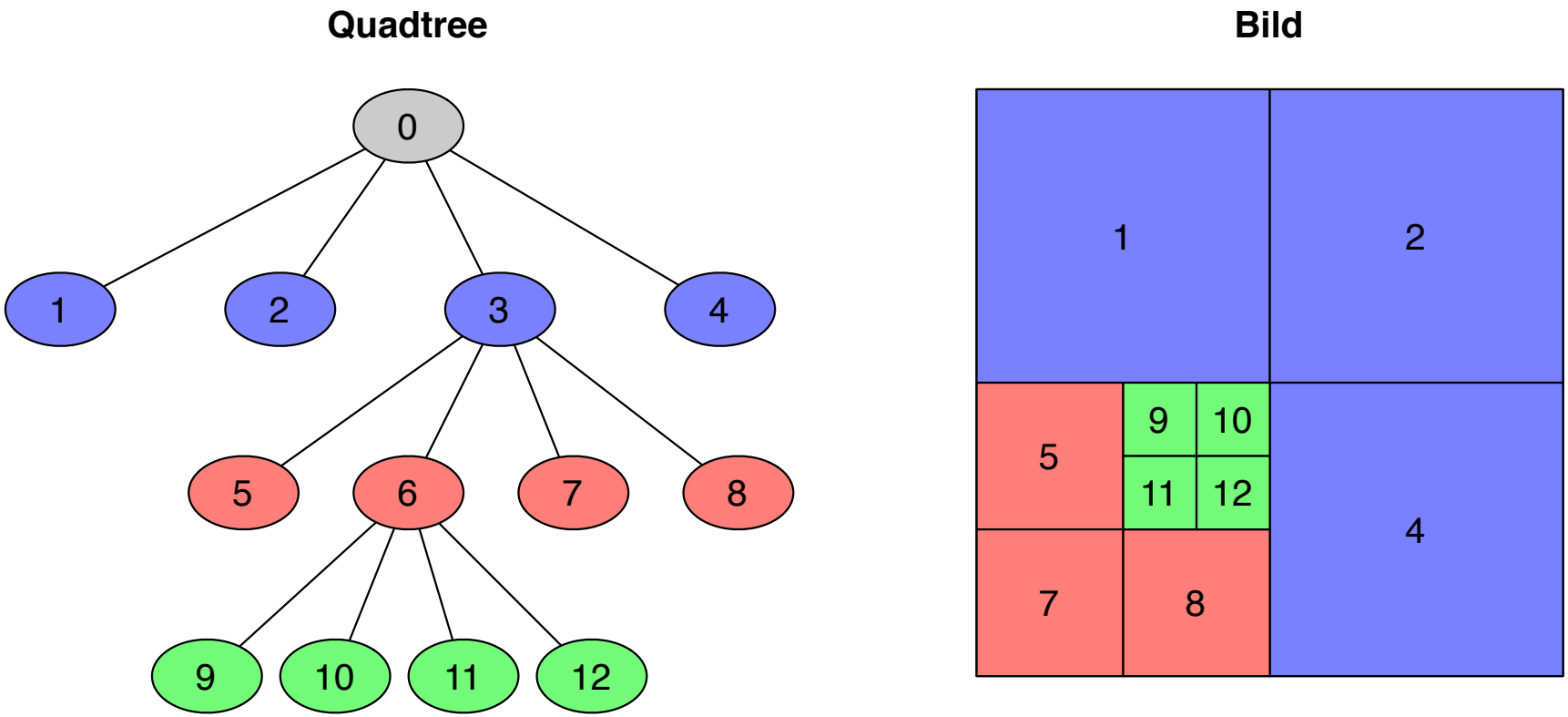
- Datenstruktur:

- Hilfsmittel: Queue von Knoten "knotenQueue"

```
breitensuche(wurzelKnoten):
    knotenQueue = leer;
    knotenQueue.enqueue(wurzelKnoten);
    while ( !knotenQueue.isEmpty() ) {
        knoten = knotenQueue.dequeue();
        besuche(knoten);
        if (knoten.links != null)
            knotenQueue.enqueue(knoten.links);
        if (knoten.rechts != null)
            knotenQueue.enqueue(knoten.rechts);
    }
```

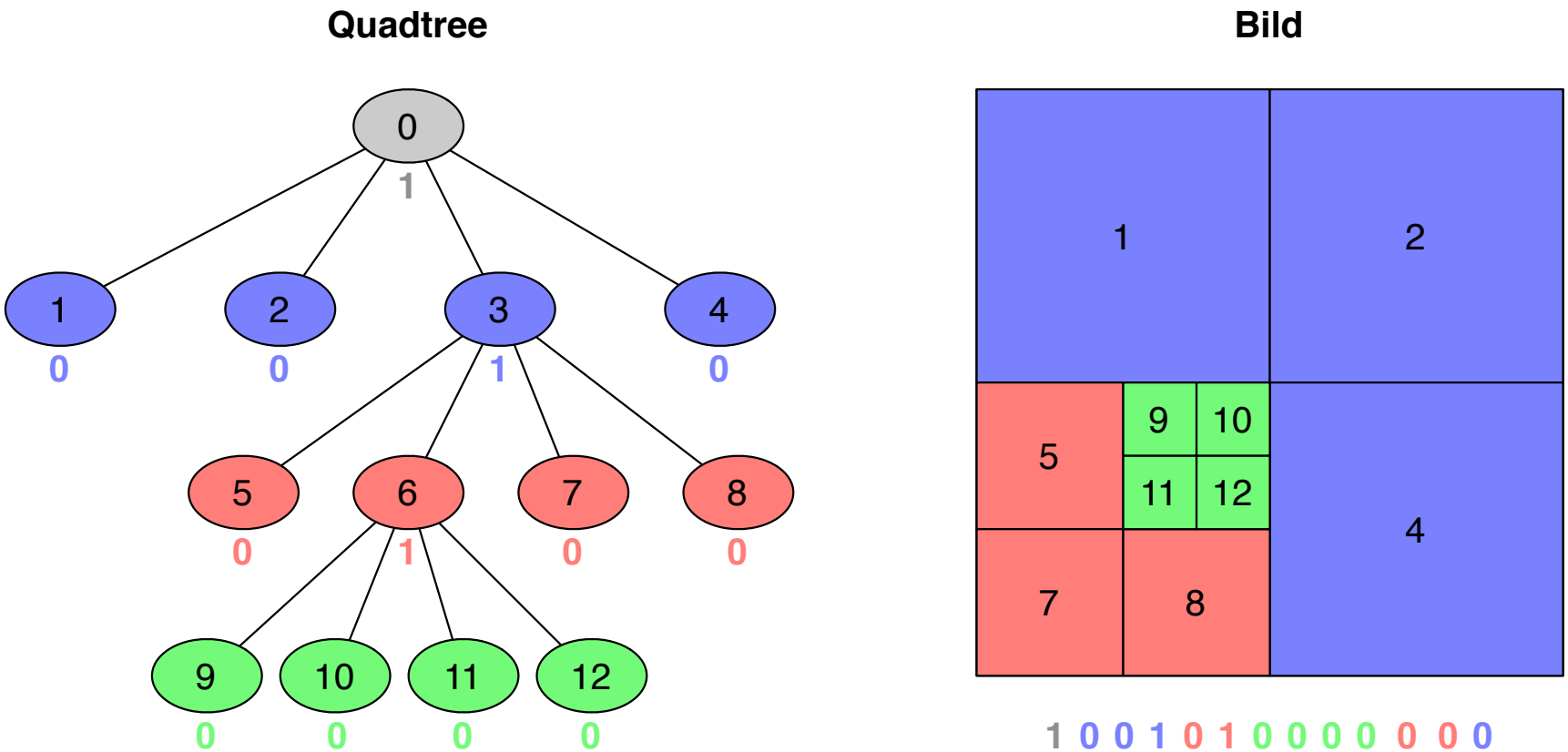
Anwendung: Quadtree I

- 4-närer Baum heißt Quadtree



Anwendung: Quadtree II

- Codierung des Baumes mit Binärziffern



Programm heute

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

Definition Heap

Definition Heap

Sei $G = (V, E)$ ein Binärbaum mit Wurzel $w \in V$. Jeder Knoten $v \in V$ sei mit einem Wert $key(v)$ verknüpft, die Werte seien durch $\leq, \geq$ geordnet.
G heißt Heap, falls er folgende zwei Eigenschaften erfüllt:

- G ist fast vollständig, d.h. alle Ebenen sind vollständig gefüllt, ausser auf der untersten Ebene, die von links her nur bis zu einem bestimmten Punkt gefüllt sein muss.
- G erfüllt die Min-Heap-Eigenschaft bzw. die Max-Heap-Eigenschaft, d.h. für alle Knoten $v \in V$, $v \neq w$ gilt

  - Min-Heap: $key(v.vater) \leq key(v)$
  - Max-Heap: $key(v.vater) \geq key(v)$

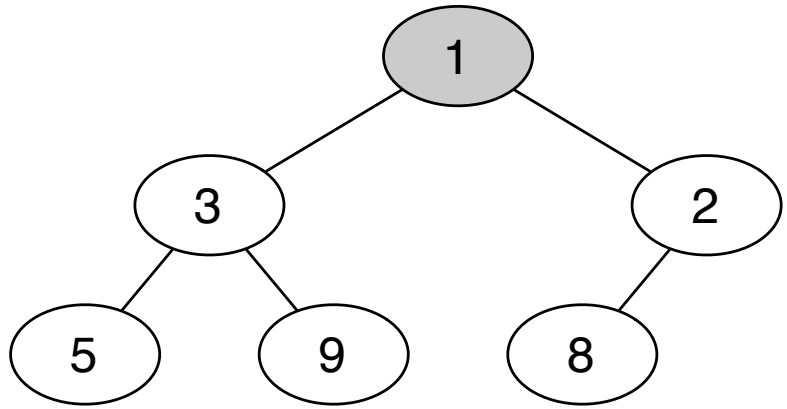
Entsprechend der Heap-Eigenschaft heißt G Min-Heap bzw. Max-Heap.

Bemerkungen zur Heap Definition

Sei $G = (V, E)$ Min-Heap.

- wir beschränken uns hier auf Min-Heaps, alle Aussagen gelten mit entsprechenden trivialen Änderungen auch für Max-Heaps
- typische Keys von Knoten sind Zahlen, z.B. $key : V \to \mathbb{R}$
  - weiteres Beispiel: Strings als Keys, lexikographisch geordnet
- ein Heap ist kein abstrakter Datentyp!

Min-Heap: Beispiel



- Keys sind hier natürliche Zahlen
- Baum ist fast vollständiger Binärbaum
- Baum erfüllt Min-Heap-Eigenschaft:
  - für alle Knoten v (ausser Wurzel) gilt

$$key(v.vater) \leq key(v)$$

Heap Eigenschaften

Sei $G = (V, E)$ Heap mit Wurzel $w \in V$.

- G als Min-Heap bzw. Max-Heap hat immer Element mit kleinstem bzw. größtem Key in Wurzel w
- G ist aber nicht vollständig sortiert (d.h. Traversierung liefert nicht notwendigerweise vollständig sortierte Folge)
- Ist $|V| = n$, so hat G Höhe von $\Theta(\log n)$

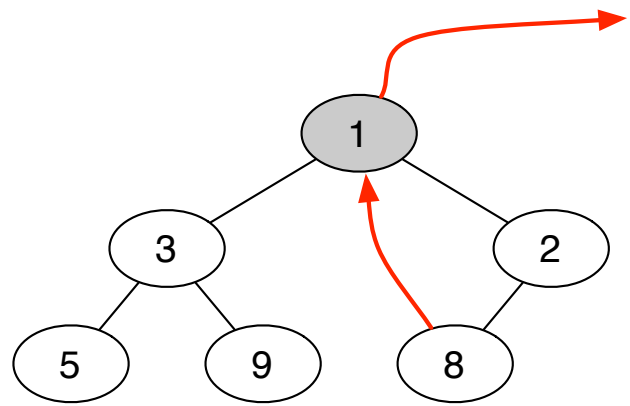
- typische Operation: extrahiere kleinsten (bzw. größten) Key (d.h. Wurzel), kurz: extractMin (bzw. extractMax)
  - anschließendes Problem: Heap-Eigenschaft wiederherstellen, als Operation: minHeapify (bzw. maxHeapify)

Heap: extractMin

Sei $G = (V, E)$ Min-Heap mit Wurzel $w \in V$.

- Operation extractMin:
  - entferne Wurzel w aus Heap G und liefere $key(w)$ zurück
  - tausche letzten Knoten von G an Stelle von Wurzel
  - stelle Heap-Eigenschaft wieder her mit minHeapify

Output: minimaler Key in G
extractMin(G):
    $min = key(w)$;
    tausche Inhalt von w mit letztem
      Knoten in G;
    minHeapify(G, w);
    return min;



Heap: minHeapify

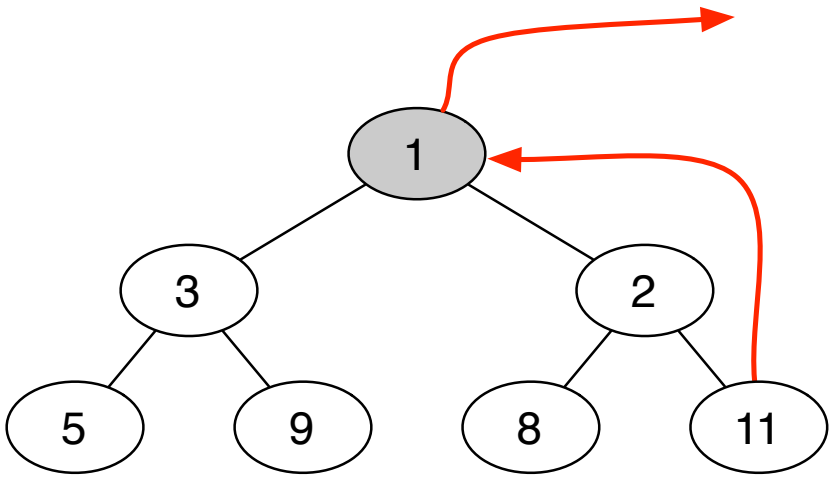
Sei $G = (V, E)$ Min-Heap mit Wurzel $w \in V$ und $|V| = n$.

- Operation minHeapify auf Knoten $v \in V$ zur Wiederherstellung der Min-Heap-Eigenschaft
- Voraussetzung: nur Knoten v verletzt Min-Heap-Eigenschaft
- lasse v durch Heap absinken, bis Min-Heap-Eigenschaft wiederhergestellt

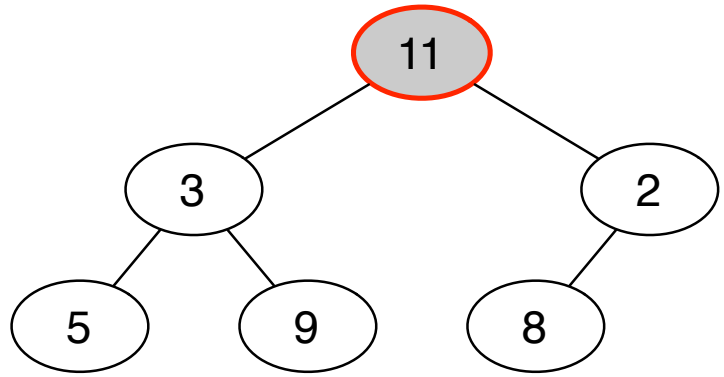
Input: Knoten v
minHeapify(G, v):
  if (v ist Blatt) **return**;
  knoten = Minimum (bzgl. key) von v.links und v.rechts;
  if ($key(knoten) < key(v)$) {
    tausche Inhalt von *knoten* und v;
    minHeapify$(G, knoten)$;
  }

- Komplexität: $O(\log n)$

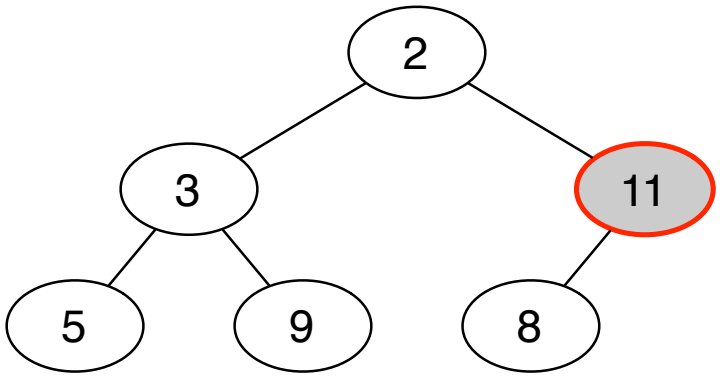
Beispiel extractMin / minHeapify



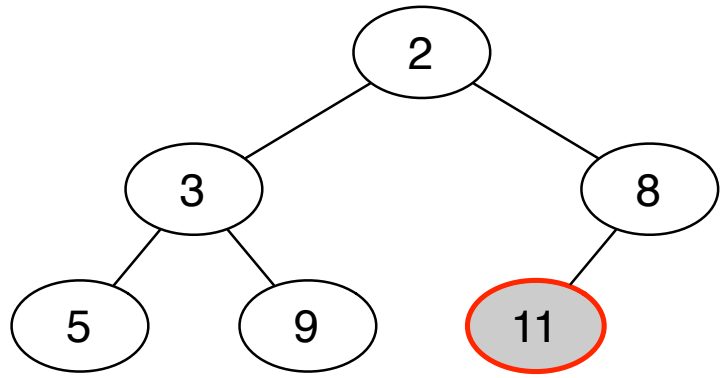
extractMin



minHeapify



minHeapify



minHeapify

Heap erzeugen: buildMinHeap

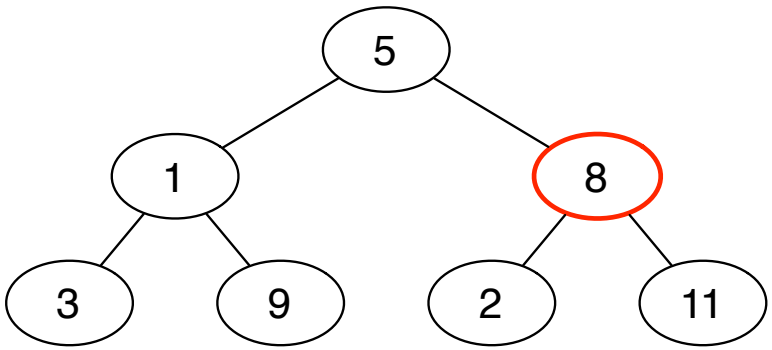
Gegeben sei Knoten-Liste V mit $|V| = n$ und Keys $key(v)$ für $v \in V$.

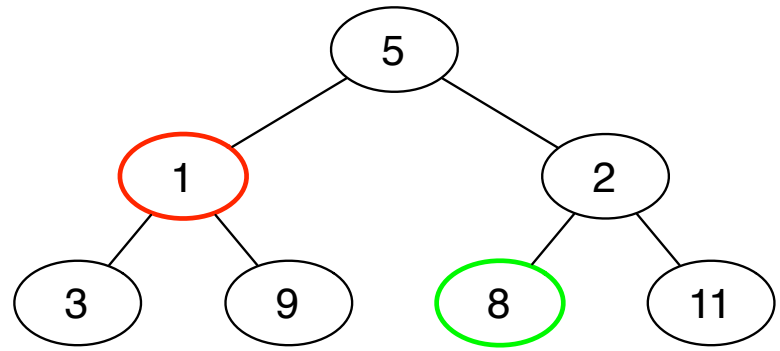
- wie erzeugt man aus V einen Min-Heap $G = (V, E)$?
  - erzeuge irgendwie fast vollständigen Binärbaum aus V
  - wende minHeapify auf alle Knoten $v \in V$ an, von unten nach oben (nicht nötig für unterste Ebene des Baumes!)

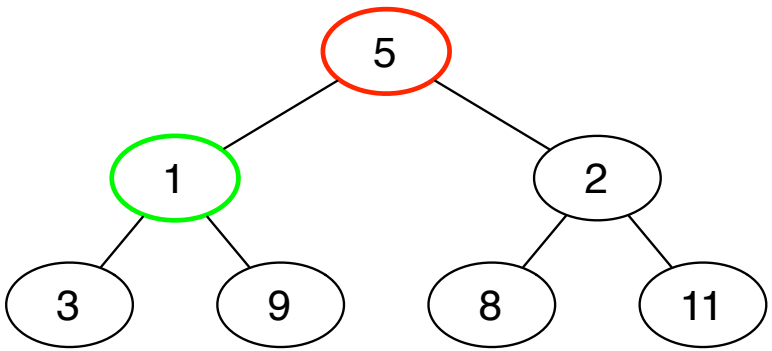
Input: Knoten-Liste V
Output: Min-Heap G
buildMinHeap(V):
  G = erzeuge beliebigen fast vollständigen Binärbaum aus V;
  for each Knoten v in G von unten nach oben {
    minHeapify(G, v);
  }

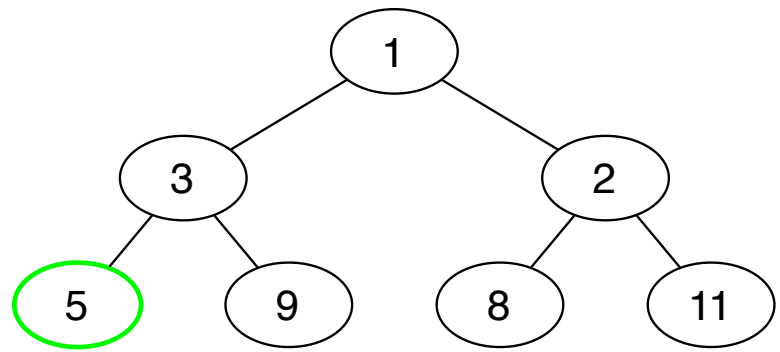
- Komplexität: $O(n)$ (nicht nur $O(n \log n)$!)

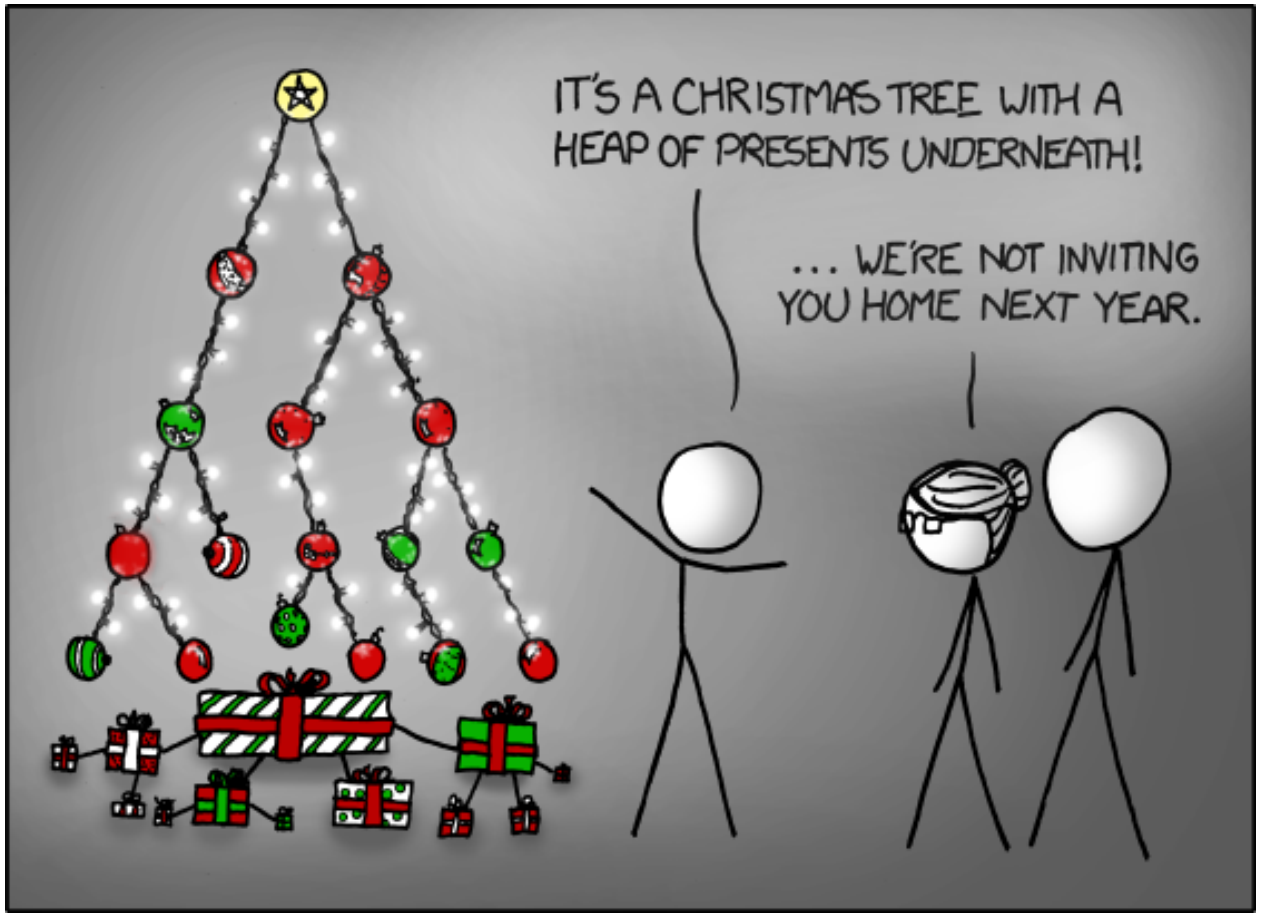
Beispiel buildMinHeap











http://xkcd.com/835/

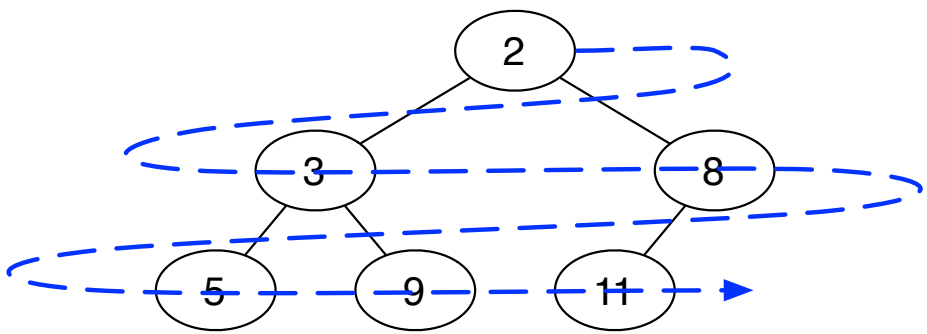
Wiederholung: Binärbaum als sequentielle Liste I

- vollständiger Binärbaum Höhe k hat $2^{k+1} - 1$ Knoten

  $\rightarrow$ speichere Knoten von oben nach unten, von links nach rechts in sequentieller Liste (Array)

  $\rightarrow$ maximale Grösse von Array: $2^{k+1} - 1$

- Beispiel fast vollständiger Binärbaum:



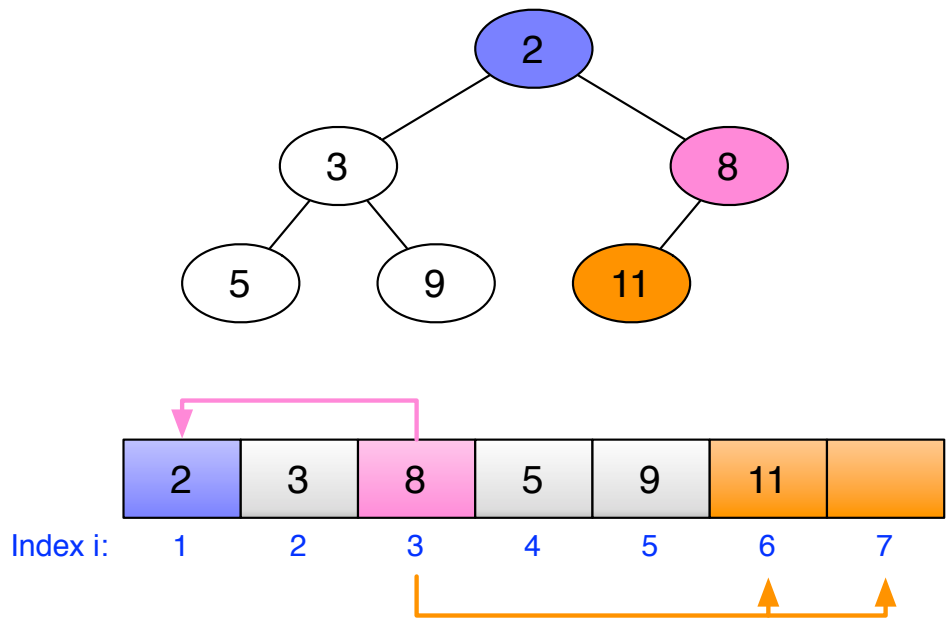
2	3	8	5	9	11	

Wiederholung: Binärbaum als sequentielle Liste II

- Wurzel: an Position 1
- Knoten an Position i:
  - Vater-Knoten an Position $\lfloor i/2 \rfloor$
  - linkes Kind an Position $2i$;
  - rechtes Kind an Position $2i + 1$

Pseudocode:

- **vater**(i): **return** $\lfloor i/2 \rfloor$;
- **links**(i): **return** $2i$;
- **rechts**(i): **return** $2i + 1$;



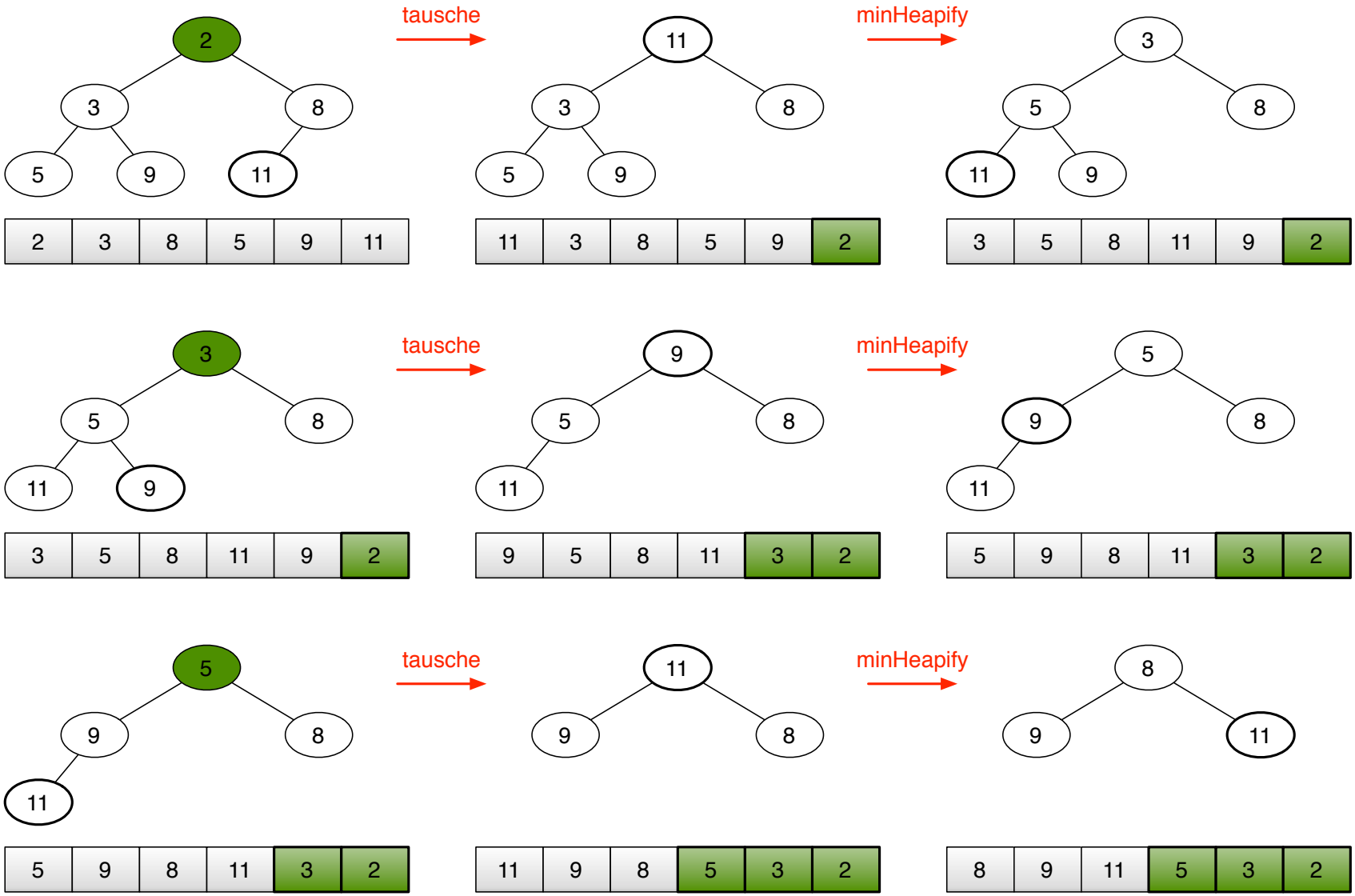
HeapSort

- Sortieren mit Heap
- Idee:
  - Heap erstellen mit **buildMinHeap**
  - wiederhole **extractMin** bis Heap leer
- mit Heap direkt im Eingabefeld:

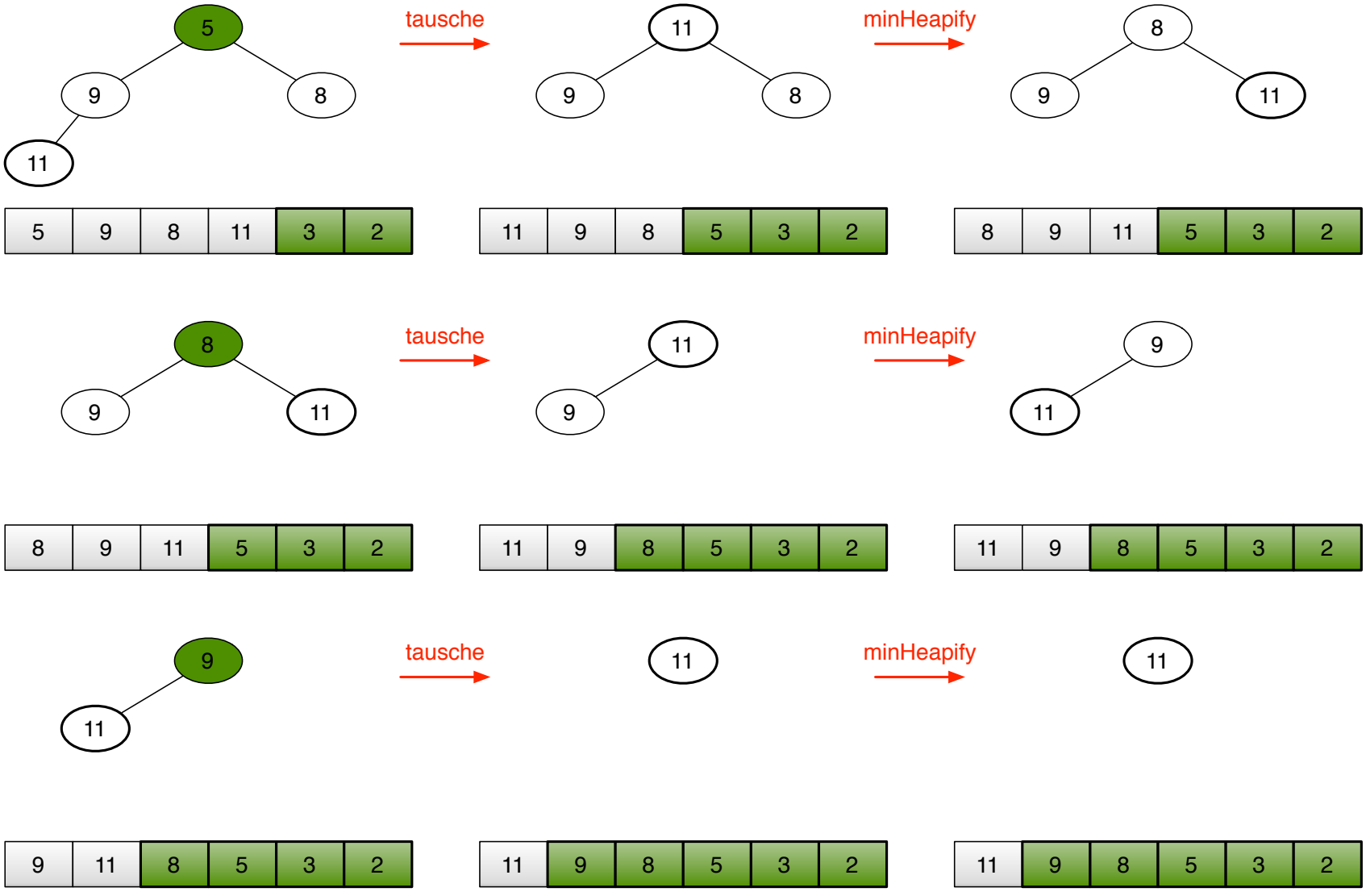
Input: Feld A[1..n] der Länge n
HeapSort(A):
    buildMinHeap(A);
    for i=n **downto** 2 {
        tausche A[1] mit A[i];
        A.length = A.length - 1;
        minHeapify(A, 1);
    }

- min-Heap sortiert in absteigender Reihenfolge
- max-Heap sortiert in aufsteigender Reihenfolge

HeapSort: Beispiel I



HeapSort: Beispiel II



HeapSort Eigenschaften

- sortiert in-place
- Komplexität $O(n \log n)$
  - besser als QuickSort im worst case!
  - in Praxis aber erst bei grossem n
- nicht stabil

Stabilität von Sortierverfahren

Stabilität

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge von gleichrangigen Elementen bewahrt.

Beispiel:

- unsortierte Liste:

  5 2 3 4 3 (blau vor rot)

- sortierte Liste (stabil):

  2 3 3 4 5 (blau vor rot)

- sortiere Liste (nicht stabil):

  2 3 3 4 5 (rot vor blau)

Sortier-Algorithmen illustriert

Animationen der Sortier-Algorithmen:

☞ `http://www.sorting-algorithms.com`

Sortier-Algorithmen Zusammenfassung

- Insertion Sort
  - in-place, stabil
  - Komplexität $O(n^2)$, best case: $O(n)$
- Selection Sort (Übung)
  - in-place, nicht stabil
  - Komplexität $O(n^2)$
- MergeSort
  - benötigt zusätzlichen Speicher, stabil
  - Komplexität $O(n \log n)$
- QuickSort
  - in-place, nicht stabil
  - Komplexität im Mittel $O(n \log n)$, worst case: $O(n^2)$
- HeapSort
  - in-place, nicht stabil
  - Komplexität $O(n \log n)$

Programm heute

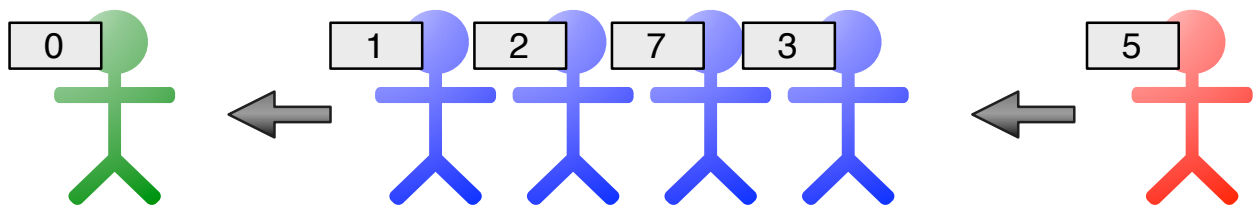
7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues

Definition Priority Queue

Definition Priority Queue

Eine Priority Queue ist ein abstrakter Datentyp. Sie beschreibt einen Queue-artigen Datentyp für eine Menge von Elementen mit zugeordnetem Schlüssel und unterstützt die Operationen

- Einfügen von Element mit Schlüssel in die Queue,
- Entfernen von Element mit minimalem Schlüssel aus der Queue,
- Ansehen des Elementes mit minimalem Schlüssel in der Queue.



- entsprechend gibt es auch eine Priority Queue mit Entfernen/Ansehen von Elememt mit maximalem Schlüssel

Definition Priority Queue (abstrakter)

Priority Queue P ist ein abstrakter Datentyp mit Operationen

- insert(P, x) wobei x ein Element
- extractMin(P) liefert ein Element
- minimum(P) liefert ein Element
- isEmpty(P) liefert `true` or `false`
- initialize liefert eine Priority Queue Instanz

und mit Bedingungen

- isEmpty(initialize()) == `true`
- isEmpty(insert(P, x)) == `false`
- minimum(initialize()) ist nicht erlaubt (Fehler)
- extractMin(initialize()) ist nicht erlaubt (Fehler)

(Fortsetzung nächste Folie)

Definition Priority Queue (abstrakter)

Fortsetzung Bedingungen Priority Queue P:

- minimum(insert(P, x)) liefert zurück
  - falls P == initialize(), dann x
  - sonst: min(x, minimum(P))
- extractMin(insert(P, x))
  - falls x == minimum(insert(P, x)), dann liefert es x zurück und hinterlässt P im Originalzustand
  - sonst liefert es extractMin(P) zurück und hinterlässt P im Zustand insert(extractMin(P), x)

(entsprechend für die Priority Queue mit maximalem Schlüssel)

Priority Queue: Implementationen I

- mit sortierten Feldern (als sequentielle oder verkette Liste)
  - insert legt Element an richtiger Stelle in sortierter Liste ab mit $O(n)$ Komplexität
  - minimum, extractMin als $O(1)$ Operation
- mit unsortierten Feldern (als sequentielle oder verkettete Liste)
  - insert hängt Element einfach an Ende an mit $O(1)$
  - minimum, extractMin suchen nach Element mit kleinstem Schlüssel mit $O(n)$

$\rightarrow$ beides nicht sonderlich effizient (je nach Abfolge der Operationen aber ok)

Priority Queue: Implementationen II

Priority Queue P als min-Heap $G = (V, E)$ mit Wurzel w:

- minimum von P liefert Wurzel w zurück
  - Komplexität $O(1)$
- extractMin von P entspricht extractMin von G
  - Komplexität $O(\log n)$
- insert von P erfordert ein klein wenig Extra-Aufwand:

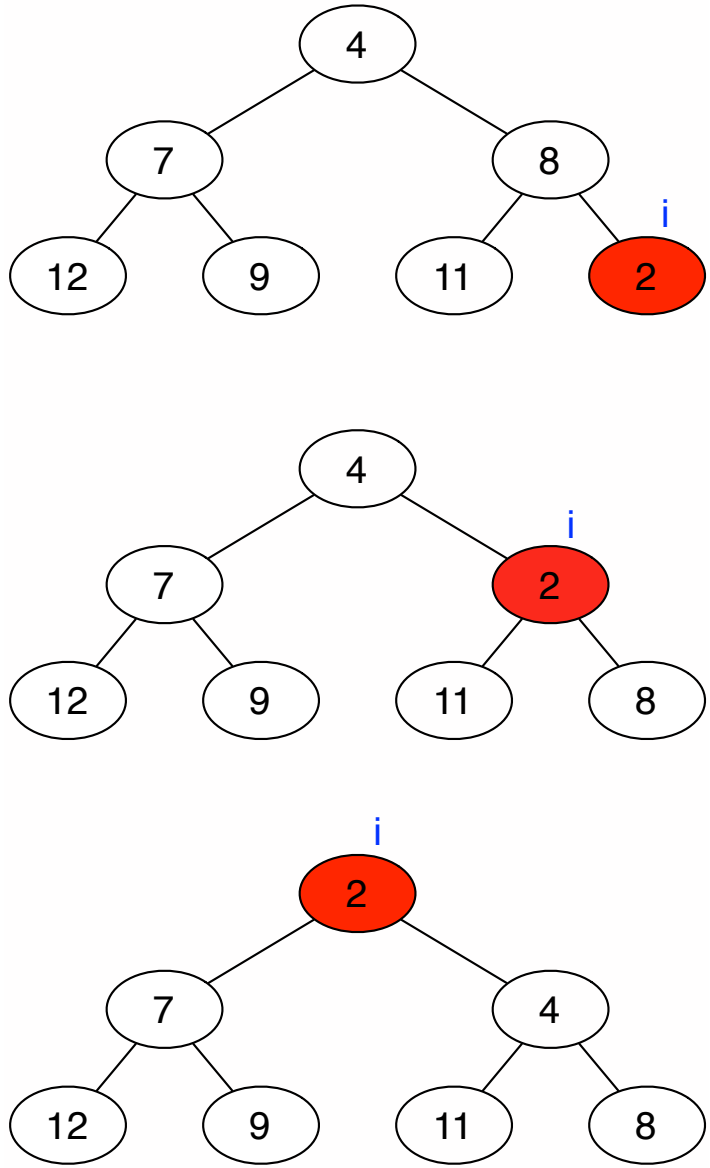
  Input: Priority Queue P (als min-Heap mit seq. Liste A),
  Element x

  insert(A, x):

  füge Element x an Ende von Heap A ein;
  i = Index von letztem Element;
  while ((i != 1) && (A[**vater**(i)] > A[i])) {
  tausche A[i] mit A[**vater**(i)];
  i = **vater**(i);
  }

  - Komplexität $O(\log n)$

Priority Queue: insert Beispiel



Priority Queue: dynamisches Anpassen von Keys

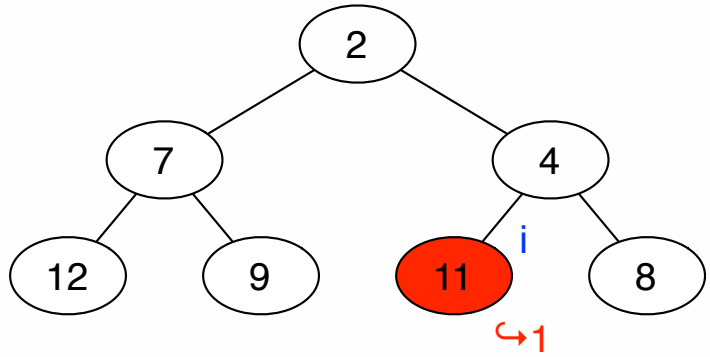
- manchmal ändert sich der "Priorität" von Schlüsseln
  - Beispiel Algorithmen dafür in Kapitel 9!
- Operation decreaseKey verringert Schlüssel von bestimmten Element

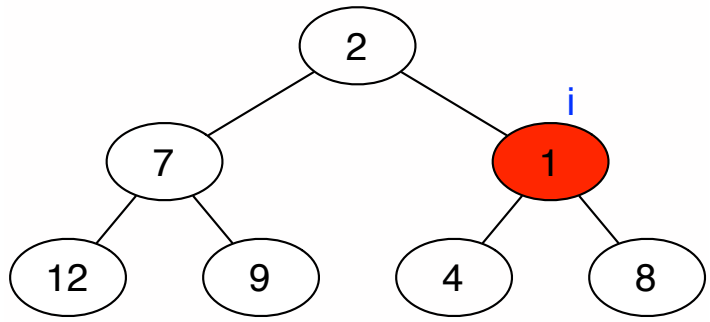
Input: Priority Queue P (als min-Heap mit seq. Liste A),
Element mit Index i, neuer Schlüsselwert wert

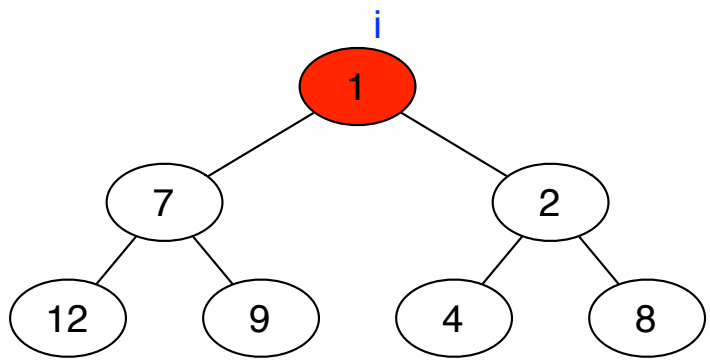
decreaseKey(A, i, wert):
  if (wert $>$ A[i]) **error** "neuer Schlüssel größer als alter!"
  A[i] $=$ wert;
  while ((i != 1) && (A[**vater**(i)] $>$ A[i])) {
    tausche A[i] mit A[**vater**(i)];
    i $=$ **vater**(i);
  }

- Komplexität $O(\log n)$

Priority Queue: decreaseKey Beispiel







Priority Queue: decreaseKey / insert

- mit Operation decreaseKey läßt sich insert anders formulieren:

  Input: Priority Queue P (als min-Heap mit seq. Liste A),
  Element x

  insert(A, x):
  A.length = A.length + 1;
  A[A.length] = ∞;
  decreaseKey(A, A.length, x);

Priority Queue: Ausblick

- Priority Queue mit Heap: insert und decreaseKey sind $O(\log n)$
- dies läßt sich mit Fibonacci-Heap bzw. Radix-Heap verbessern auf (amortisiert) $O(1)$

  $\rightarrow$ Effiziente Algorithmen in Informatik

Priority Queues und Sortieren

- mit Priority Queues lassen sich Sortier-Algorithmen implementieren
- Schema:
  - alle Elemente in Priority Queue einfügen
  - der Reihe nach alle Elemente mit extractMin / extractMax entfernen
- Beispiele:
  - Priority Queue mit Heap: HeapSort
  - Priority Queue mit sortierter sequentieller Liste: Insertion Sort
  - Priority Queue mit unsortierter sequentieller Liste: Selection Sort

Zusammenfassung

7. Fortgeschrittene Datenstrukturen
   - Graphen
   - Bäume
   - Heaps
   - Priority Queues